

# LYNX COMPUTING



IAN SINCLAIR











# **Lynx Computing**





# **Lynx Computing**

**Ian Sinclair**

**GRANADA**

London Toronto Sydney New York

Granada Publishing Limited – Technical Books Division  
Frogmore, St Albans, Herts AL2 2NF  
and  
36 Golden Square, London W1R 4AH  
515 Madison Avenue, New York, NY 10022, USA  
117 York Street, Sydney, NSW 2000, Australia  
60 International Boulevard, Rexdale, Ontario, R9W 6J2, Canada  
61 Beach Road, Auckland, New Zealand

Copyright ©1983 by Ian Sinclair

*British Library Cataloguing in Publication Data*

Sinclair, Ian, R.

Lynx computing.

1. Lynx (Computer)

I. Title

001.64'04      QA76.8.L/

ISBN 0-246-12131-9

First published in Great Britain 1983 by Granada Publishing

Typeset by V & M Graphics Ltd, Aylesbury, Bucks

Printed in Great Britain by Mackays of Chatham, Kent

All rights reserved. No part of this publication may be reproduced, stored in a retrieval system, or transmitted in any form or by any means, electronic, mechanical, photocopying, recording or otherwise, without the prior permission of the publishers.

Granada ®

Granada Publishing ®



# Contents

<i>Preface</i>	vii
1 Setting Up	1
2 Ins and Outs	14
3 Counting, Listing and Looping	26
4 Numbers and More Numbers	39
5 Strings and Things	51
6 Structures, Lists and Menus	66
7 Number and String Lists	80
8 Beginning Graphics and Sound	88
9 Higher Resolution All Round	104
10 More Sound and Some Afterthoughts	120
<i>Appendix A: Cassette Loading Problems</i>	130
<i>Appendix B: ASCII Codes</i>	132
<i>Appendix C: Editing</i>	133
<i>Appendix D: Magazine and User Groups</i>	134
<i>Appendix E: Mystery Corner</i>	136
<i>Appendix F: Drawing Circles</i>	137
<i>Glossary</i>	138
<i>Index</i>	145





# Preface

As computer design progresses, the amount of computing capability that you can buy for a fixed amount of money seems to increase almost indefinitely. The other side of that particular coin is that computers become less easy to understand, particularly by the raw beginner. Even the more experienced user may find that new designs of computers incorporate instructions which are novel and which require considerable explanation. In particular, the topics of high resolution graphics, colour and sound have to be emphasised because they will be new to anyone who has not bought one of the new generation of computers. Further features of the Lynx, such as the approach to structured BASIC, have not been available generally on more than one type of machine until now.

Computer manuals vary in quality, and the Lynx manual must be counted among the best. A computer manual, however, has to contain information for all users, and it can never be so comprehensive as to contain everything that a beginner needs. This book has therefore been written with a view to initiating the beginner to this remarkable machine. Because I feel that the machine will also have a particular appeal to buyers with business, scientific, engineering or mathematical backgrounds, I have emphasised these aspects. It is, nevertheless, a machine that can be used and enjoyed by a wide variety of users. It also has the considerable advantage of being easily expanded to a very powerful computer (by any standards of microcomputing) for business purposes.

One important aspect of Lynx computing has been completely omitted, however. The machine is particularly well designed to permit the use of 'machine code' programming. For the beginner, however, this type of programming, which requires considerable knowledge of how the machine works, is considerably more difficult. Since a reasonably full treatment of machine code

programming would require a book, larger than this, devoted to that one topic, I have omitted it completely on the grounds that a little learning would be a dangerous thing. It should also be pointed out that some earlier models of Lynx did exhibit some arithmetical inconsistencies but these should not appear on the production machine.

A book like this is always the result of much hard work by a great number of people. As always, Richard Miles, of Granada Publishing Ltd, has been continually helpful in coaxing manufacturers, editing, proofing, and generally whipping the book into readable order. The Computers team in Cambridge, particularly Davis Jansons, are greatly to be thanked for their easy approachability (would that all manufacturers were like that!), and willingness to discuss aspects of the machine at length. I am also very grateful to Geoff Sore for undertaking to print out the listings for this book. At the time when I wrote this, it was impossible to obtain an interface to connect the Lynx to my Epson MX-80, and the Computers' Seikosha GP-250 saved the day.

I am also greatly indebted to Jo and Sally Lang, of Lang Communications for the efforts that they made to have the Lynx delivered to me by Christmas Eve, and for the many fruitful conversations that I had with them on the subject of Lynx computing.

Finally, I dedicate this book to Morag, who identifies strongly with all creatures feline.

Ian Sinclair

# Chapter One

## Setting Up

A computer, unlike a kettle or a toaster, can't just be plugged in and switched on. It's a complicated machine which you need to understand to some extent if you are to be able to get the best from it. By the time you finish this book, you will know the program capabilities of your Lynx, using the BASIC programming language, very well indeed. We'll start, however, in a very modest way. Many buyers of the Lynx will have moved on to this machine from older designs; other buyers will undoubtedly be first-time computer owners. If you have used a computer before, the operation of the Lynx will be less mysterious, but that doesn't mean that there will be no surprises. The design of computers has moved on so much in the last three years that you may find the capabilities of the Lynx so far advanced compared to your old computer that you don't know where to start. This book will guide you. If, on the other hand, you are a first-timer, attracted by the considerable capabilities of the Lynx, then this book will be ideally suited to you. I shall start from scratch, assuming no previous knowledge of computers, or of anything else electronic.

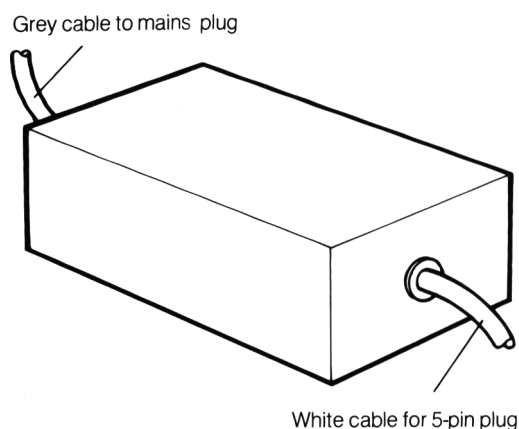
To see what the Lynx is doing, we need to connect it to a TV receiver or to a monitor. A monitor is a type of superior grade TV display which is designed to make use of the signals from computers or video recorders directly, rather than transmitted signals picked up with an aerial. A good quality colour monitor, like the Microvitec, can give colour pictures of a quality that you are not likely to have experienced using a TV receiver, but this quality of picture is not necessary in order to learn computing. You could learn a lot about the Lynx using only a simple black/white portable receiver such as the Ferguson which I used when writing the programs that are used in this book. Obviously, when we come to look at the colour commands of the Lynx, it is an advantage to be able to use a colour receiver or monitor to see the program working.

## 2 Lynx Computing

It's not essential, though, because the colours appear on a black/white receiver as shades of grey.

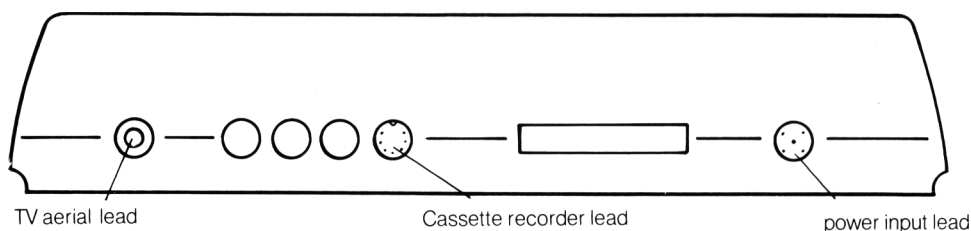
### Mains supplies

Down to work, then. The Lynx does not work directly from the high mains voltage, but from a number of lower voltages. These are supplied from a 'power pack' – the metal box that comes along with the Lynx (Fig. 1.1). There are two cables emerging from this box.



*Fig. 1.1.* The 'power pack' of the Lynx.

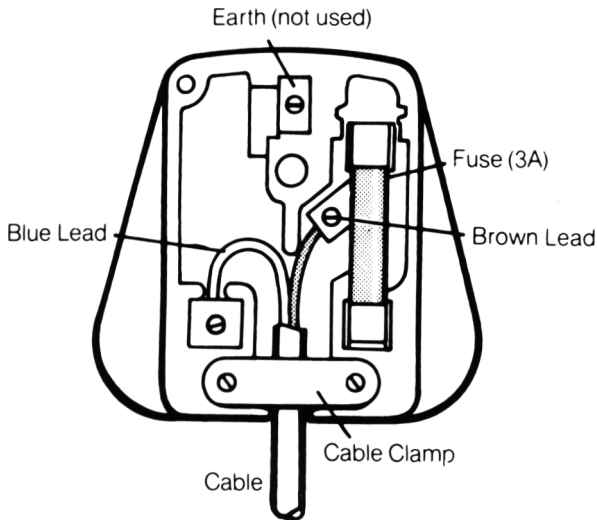
The white cable carries a small four-pin plug which engages into a socket at the back of the Lynx, on the right-hand side as you look at the back view (Fig. 1.2). Take a close look at this plug and socket.



*Fig. 1.2.* The tail view of the Lynx, showing the TV, cassette and power sockets.

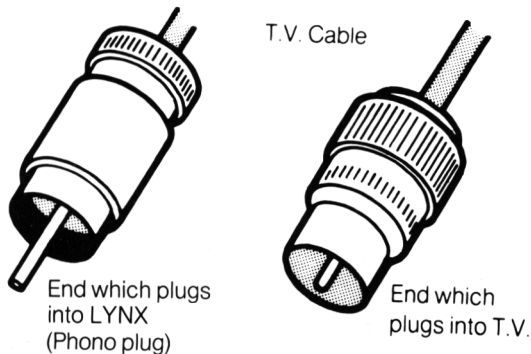
You will see that both are shaped so that the plug will go in one way round. It's important not to force it, because if you manage to force it in the wrong way round and then switch on, you can do a lot of damage to the circuits of the Lynx. The plug should slip in

reasonably easily with a firm push and a little bit of wiggling (the plug, that is). Once in place, it should not be loose – a loose connection here will cause all sorts of peculiar behaviour.



*Fig. 1.3.* Connections to a three-pin plug. Don't try it unless you have electrical experience. The fuse must be a 3A type.

The other, longer, grey cable from the power pack is the mains lead, and you will need to fit a three-pin mains plug to it. If you are familiar with fitting mains plugs, Figure 1.3 is a reminder of the connections. You must use a 3A fuse rather than the more usual 13A



*Fig. 1.4.* TV cable and plugs. The plugs are not interchangeable.

size. A computer needs very little power, and a large fuse rating, such as 13A, is undesirable on safety grounds. If you are in any way

## 4 Lynx Computing

uncertain about fitting the plug, get your local electrical shop to do this job. You don't have to take the Lynx to the shop, only the power pack.

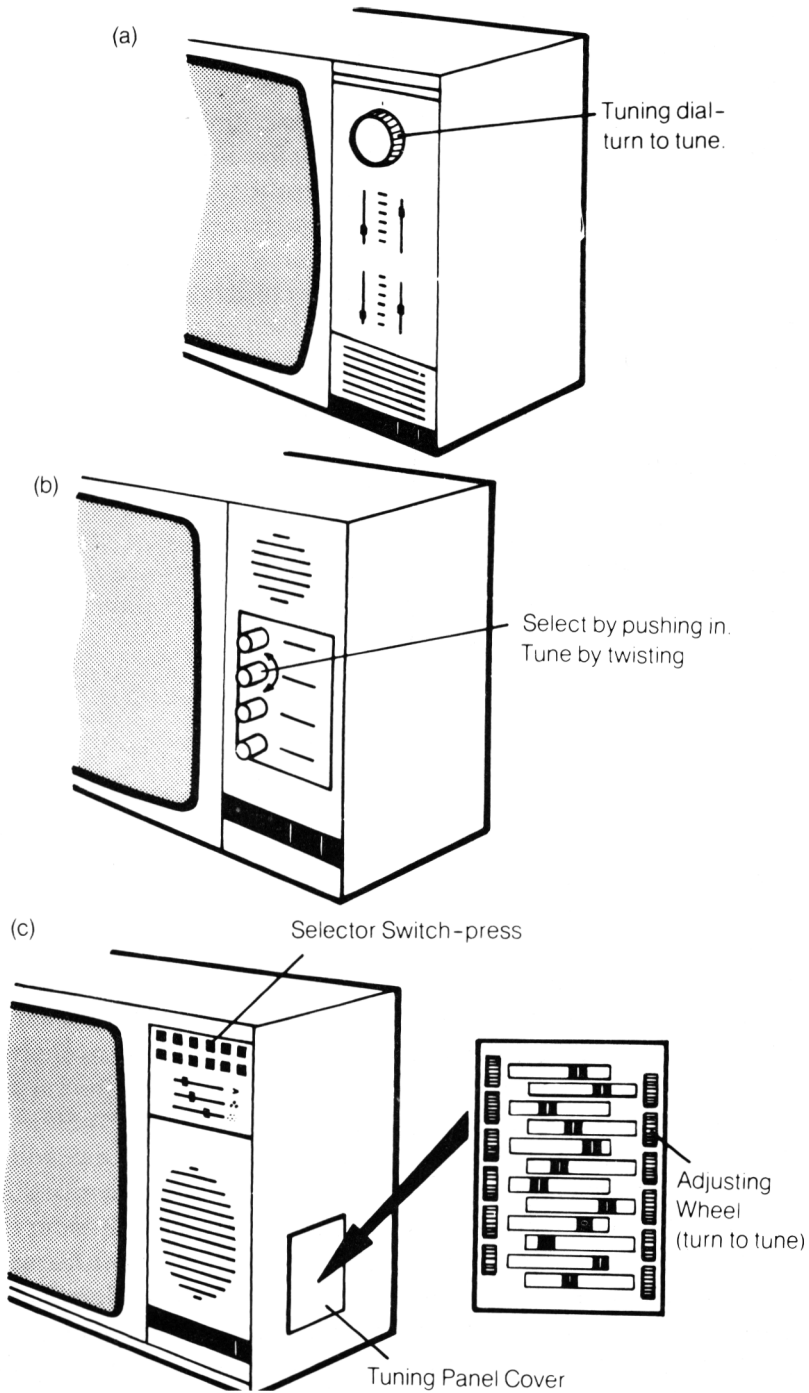
Note that Lynx does not have an on/off switch. You should always pull out the mains plug after a computing session, or switch off at the mains socket. Don't pull out the four-pin plug that connects the Lynx to the power pack – that doesn't switch the power pack off, and you may have problems with the Lynx if you try to switch it on by plugging this connector in. In addition, frequent plugging and unplugging of this connector can cause it to work loose.

### TV connections

Plugging the Lynx into its power pack, and the power pack to the mains will start the Lynx operating, but there is no visible sign of this. To see what the Lynx is doing, you need to connect it to the TV. Assuming for the moment that you are using a TV receiver rather than a monitor, this is done using the cable that is supplied with the Lynx. This cable (Fig. 1.4) has differently shaped plugs at its two ends. The plug with the projecting contact is pushed into the TV socket on the back of the Lynx (see Fig. 1.2 again), and the plug with the recessed contact fits into the aerial socket of the TV receiver. So far, so good.

Now you can switch on the TV and let it warm up. Switch on the Lynx also. You will hear the loudspeaker in the Lynx make a short 'beep' sound as the machine is switched on. Unfortunately, it's likely that you won't see anything on the screen. The reason is that a TV receiver is designed to receive signals on a number of different 'channels'. Unless the TV happens to be adjusted, *tuned* as we call it, to receive the channel that the Lynx uses to transmit its signal, you'll see nothing. To find the Lynx signal, you will have to re-tune the TV. The method that you have to follow to do this depends on the type of TV that you are using.

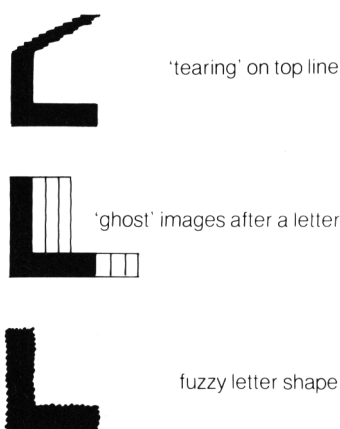
Figure 1.5 shows the three most common varieties of TV tuning methods. Most B/W portables use dial tuning, in which a knob or rim of a dial can be rotated to receive any of the normal TV channels. This type of tuning is illustrated in Fig. 1.5(a), and you can tune in the Lynx signal simply by turning the dial until the tuning signal appears. The signal is correctly tuned when the word Lynx and the paw-mark symbol appear on the screen clearly. Figure 1.6 shows



**Fig. 1.5.** TV tuning methods. (a) Dial system, (b) type using large push-buttons, (c) modern type using small push buttons or touch pads, with a separate tuning panel.

some of the defects that need to be remedied by more careful tuning.

If you are using a colour TV receiver, it's more likely that it will be fitted with some form of push-button or touch-tuning system. Large B/W receivers also may be fitted with these tuning methods. When push-buttons are fitted (Fig. 1.5(b)), tuning is carried out first by selecting a button which you will use for the computer signals. In a set of four buttons, as fitted to older sets, the usual one to use is the number 4 button (sorry about that, Channel 4). If more than four buttons are fitted, use the one that carries the highest number. Push this button in until it locks in place, and then rotate it. Rotating the button retunes the receiver, and you will probably have to rotate the button through several revolutions. If you can't see the Lynx name appearing on the screen as you rotate, and you then reach the end of travel in one direction, you will then have to turn the button in the opposite direction. Once again, you should end up with the Lynx message on screen, free of the defects that are listed in Fig. 1.6. Once this message appears, small adjustments can make a considerable difference to the appearance of the picture.



*Fig. 1.6.* Defects which indicate incorrect tuning.

Modern receivers use mainly touch-pads or small push-switches, and the tuning controls are set behind a panel at the front or at the side of the receiver. Push the button or pad that you want to use with the Lynx signals, and then find the tuning controls behind the panel. Select the tuning wheel or knob which carries the same number as the button which you selected, and turn the tuning wheel until you have found the Lynx message. As before, if you find that you have turned the tuning wheel or knob all the way in one direction, then



you will have to start turning in the other direction. The tuning for a colour TV is usually more fussy – in particular, you have to tune it so that there are no coloured fringes round the letters of the word Lynx.

Once you have gone through all the effort of tuning a TV to take the Lynx signals, it makes sense not to have to change it. It also makes sense not to have to keep plugging and unplugging the aerial leads. You may find one of the 2-to-1 TV aerial adaptors (Fig. 1.7), which are sold in electrical shops, are very useful. This way, the Lynx can share the domestic TV with Coronation Street – but not at the same time!

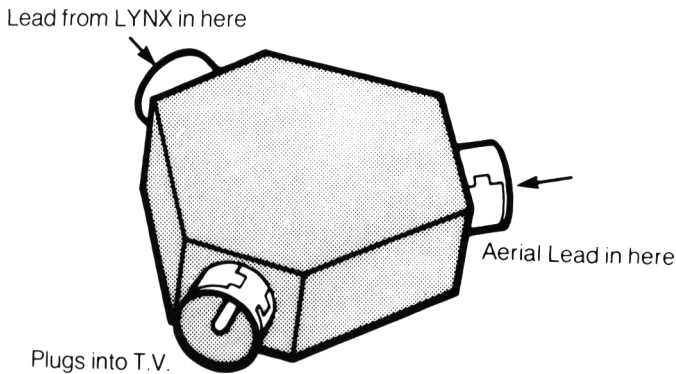


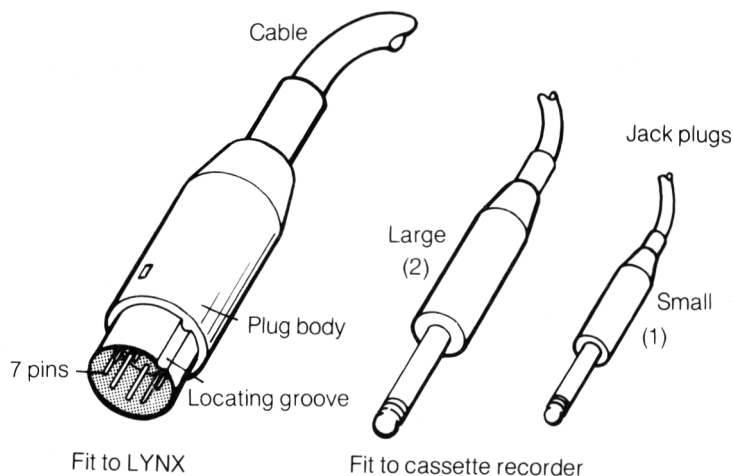
Fig. 1.7. A typical 2-to-1 TV adaptor.

### Don't rush!

It's very tempting, when you have a working computer in front of you, to start pressing every key in sight to see what happens. By all means do so if it makes you feel better, but when you have finished, switch off, wait, and then switch on again. You won't do any damage to the computer by pressing keys, but you can make it difficult to regain control over the computer so that you can start programming in earnest.

You can always recover from any peculiar effects that are caused by random key-pressing by switching off and starting again. When you switch off a computer, though, you will always lose any program that was stored in the memory of the computer. The next step, then, is to learn how to record programs on a cassette and how to replay them from the cassette. If you are thoroughly familiar with how to do these things by the end of this chapter, then you can record the programs in this book as you work through them. In this way it will be easier for you to find out what each one does later when

you have forgotten. Don't forget, incidentally, that though key-pressing can't damage your computer, plug-pushing can. Never connect or disconnect anything while the computer is switched on – always switch off first.



*Fig. 1.8.* The plugs that are fitted to the cassette lead.

Switch off now, and connect the cassette recorder. The cassette plug for the Lynx has seven pins, and it plugs into the socket on the back of the Lynx (Fig. 1.8). At the other end of this cable are three small plugs, one of which is noticeably smaller than the other two. These are called jack plugs, and the smaller one is for motor control of the cassette recorder. The two larger jack plugs are for recording and replaying signals. You can use virtually any cheap portable cassette recorder, but the best-suited machines will feature automatic recording level control, remote motor control, and a tape counter. If you are buying a cassette recorder for the Lynx, then these features should be considered as essential. If you already have a recorder that lacks one of these features (usually the tape counter), then you can get by. I used the Trophy CR-100, but there are several models in the £20 to £30 price range which are equally suitable.

You will also need cassettes. For keeping your favourite programs, the special short lengths of tape on cassette, stocked by computer shops and several High Street chain-stores, are ideal. For recording your own programs as they are developed, it's useful to have longer tapes, and I use Sony C90 for that purpose. Tapes by Agfa, Maxell, Scotch, BASF and Yashima have also proved reliable when I used them. Beware of bargain offers of unknown brands,

though; they are usually quite unsuitable for computing use.

Suppose, then, that you have a recorder of the type that is best suited to computing. It will have several sockets – three in particular that are marked MIC, EAR and REM. The MIC socket is for a microphone lead, and one of the larger Lynx plugs goes in here. The EAR socket is for earphones – it's sometimes marked with a drawing of an ear – and another Lynx plug goes in there. The smaller socket marked REM is for remote control of the tape motor, and it accepts the smaller plug of the set.

Now the trouble is that you can't be too sure which of the larger plugs goes into which of the sockets. On my Lynx, there was a grey spot on one lead. This marked the lead of the plug that fitted into the MIC socket of the recorder. The procedure I shall now describe lets you check which plug should be placed in which socket – and another vital feature!

Plug in the remote control plug, ignoring the other two for the moment. Now switch on the Lynx and the TV. You should always use the cassette recorder on its mains supply, and you'll probably need a three- or four-socket mains lead, sold in all electrical shops, to cope with all the plugs. Don't use the old-style adaptors – their contacts are never reliable enough. Now press the PLAY key of the recorder, and look at the spindles. Neither of them should move continually (although a slight twitch is acceptable, at the instant when you switch on). If you see one spindle moving, then the leads to the remote plug need to be reversed. This is a tricky job unless you are fairly good with a miniature soldering iron, so I suggest a trip to the nearest good radio *repairer* (not all shops that sell them can repair them!) or computer store to work on the leads. Later models of the Lynx will not need this modification, which in any case may not be necessary – it depends on which cassette recorder you use. It was necessary for my Trophy recorder. When all is well, it should be possible to press the PLAY key on the recorder and see nothing happen.

Now we can test recording. Place one of the larger jack plugs into the MIC socket. On my leads, this one was identified by a grey dot, but yours may not carry this identification. Now take a blank cassette and hold it with the 'A' or 'I' side uppermost. The tape that you can see at the edge of the cassette is *not* recording tape – it is a piece of plain plastic tape called a 'leader'. You can't record on to this piece of tape, so it has to be wound on to the reel before you start recording. If your recorder has a tape counter, put the cassette into the recorder with the 'A' or 'I' side uppermost. Reset the tape

counter to zero. Now pull out the remote plug, the smaller jack plug, and fast-wind the tape until the counter reading is 5. If you have no tape counter on your recorder, then use a BiC pen instead of the recorder. Insert the body of the pen into the centre of the empty spool of tape, and wind the tape until you see the plastic leader replaced by the brown coating of the recording tape. Then replace the cassette in the recorder.

With the recorder all ready to go, you need something to record. This means a program, and at this stage the simplest possible program will do. Figure 1.9 illustrates what is wanted – you have to type it.

---

```

1Ø REM
2Ø REM
3Ø REM

```

---

*Fig. 1.9.* A simple program to test recording and replaying on your recorder.

Start by typing the number 1Ø (yes, 1 and then Ø, just like a typewriter). Then type the letters REM. It doesn't mean 'remote control' to the computer, but REMinder. The number means that this is a program instruction, one of a numbered set that will be obeyed in sequence. When you have typed the M of REM, press the key that is marked RETURN. This is the key at the extreme right-hand side of the lowest row of keys, the row that starts ZXC ...

As you type the letters, the flashing block which is called the cursor shows where the next letter or number will be placed on the screen. If you hold any key down, its action will start to repeat, so just touch the keys lightly. If you press the DEL (DELeTe) key, the cursor will step back, wiping out the letter or number that it replaces. If you keep the DEL key pressed, it will wipe out a whole line. After each 'M' of REM, press the RETURN key. This action signals to the computer that you have completed this instruction, and that the computer must prepare to store another one. Your program should now look on the screen as shown in Fig. 1.9.

Now this program doesn't do anything, but it can be recorded. Type SAVE"". The letters are straightforward, and the quotation marks (or quotes) are on the key that also carries the figure 2. Since the quotes are shown above the 2, that's a sign to you that you need to press the SHIFT key along with the 2 key to type this mark. If you already know how to type, you'll be pretty familiar with the system.

You don't need a space between the quotes. If you type any name between these quote marks, the computer will treat this as a *filename*. A filename is a kind of identification, so that the computer will be able to recognise this program again. The snag is that it will not recognise it unless you ask for the identical filename. The machine will, in fact, be able to skip past programs with other filenames to pick out the one which you have asked for. That's why we use filenames but, for this exercise, we want to dispense with a filename, so there's nothing, not even a space, between the quotes.

When you have typed SAVE`""`, make sure that there is a jack plug in the MIC socket of the recorder, and that the remote control jack is inserted. Press the RECORD and PLAY keys of the recorder (some Sharp models require you only to press the RECORD key, but this is unusual). The motor should *not* start yet. Now press the RETURN key of the Lynx. After a very short delay, just enough to make you wonder if it works, the motor of the recorder should start. Your program is being recorded – we hope. When the computer has finished sending signals to the recorder, it will switch the recorder motor off again.

You should now press the STOP key of the recorder. If you leave the keys pressed down with the motor not running, there is a danger that you will cause a flat spot to be worn on the rubber wheel that pulls the tape along. If this happens, the recorder will never be reliable again until the wheel is replaced, and just try getting spares ...! Be sure – always press the STOP key when the recorder has stopped.

Now you need to find if the program has recorded correctly – or at all. Pull out the remote control jack plug, rewind the tape, turn the volume control of the recorder half-way up, and press the PLAY key. If you hear notes with rough warbling sounds at times, all fairly loud even when the volume control is turned well down, then you have a successful recording, and the plug that you have in the MIC socket is the correct one. If there's nothing there, only some low humming noises, it indicates that the wrong plug was in the MIC socket, so try again with the other one.

When you have a recording on the tape, you can push the other plug into the EAR socket. Pull out the REMote plug, run back the tape, and type NEW on the Lynx keyboard. This will remove your program. To prove it, type CLS and press the RETURN key. The screen clears. Now type LIST, and press the RETURN key. There's nothing – because there is no program to make a list of. Now type LOAD`""`. Remember that there is no space between the quotes.

Don't press RETURN yet. Set the volume control of the recorder to half-way, and if there is a tone control, set it to the maximum treble – the shrillest sound. Press the RETURN key of Lynx and then the PLAY key on the recorder. As the tape plays, the 'filename' will be replayed first. This will cause another "" to appear on the screen. When the Lynx has found and replayed the rest of the program, it will stop the motor of the recorder. You should then press the STOP key of the recorder. To show that your program has replayed, type LIST and press the RETURN key to see the program in all its glory.

If the computer simply lets the tape run on past the position that was on the tape counter when the program was recorded, press the ESC (ESCape) key of Lynx. This should restore control – you know when you have control again when the cursor appears. Now press the STOP key of the recorder. Try another LOAD"" with a higher volume-control setting. The Lynx has an unusually good cassette system, and it's not sensitive to volume control settings. You may at some time, however, buy a Lynx program on a cassette and find that this cassette will not replay (or LOAD) its program correctly into the machine. This indicates that you need your head seen to – the head of your recorder, that is! For details, see Appendix A.

The Lynx can send information to the recorder, or play it back at five different speeds. Unless you specify otherwise, the Lynx always uses the slowest speed, which is rather a pity. If you want to use long programs, then you can cut down the time that you need for recording (SAVEing) or replaying (LOADing) by using one of the faster speeds. The best speed for this purpose is obtained by typing TAPE3, then pressing RETURN. The snag is that you have to remember to type this each time you want to use the tape. A program that was recorded using the slowest speed can't be replayed using a faster speed; the two must match exactly. It's a pity that TAPE3 is not the speed that is selected when the Lynx is switched on, because it is fast and reliable. The fastest speed, TAPE5 is very fast, but very few recorders are likely to be able to use it.

### **The keyboard**

All of this cassette work will have familiarised you to some extent with the keyboard. It's a particularly good keyboard, not one of the horrors that you sometimes get on low-priced computers. The main letter and number keys are set out just like those of a typewriter, but

with the special computer keys on each side. Of these special keys, the most important are the ESC (ESCAPE), CTRL (ConTRoL), RETURN and DEL (DELeTe) keys. Unlike the action of a typewriter, the letter keys produce capitals (upper-case) when you press them, and small letters (*lower-case*) when you press the SHIFT key at the same time as a letter key. By pressing the SHIFT LOCK key once, you can reverse this. Pressing SHIFT LOCK a second time will get you back to the normal state for computers, which is typing in *upper-case* (capitals). The importance of this is that instructions to the computer should preferably be in upper-case. The Lynx, unlike most computers, is very tolerant and will let you use either lower-case or upper-case letters for instructions. It will also catch some mistakes early on. If you type an instruction which is impossible to carry out, or which is mis-spelled, the Lynx will refuse to accept the instruction, and remind you with an error message why the instruction is unacceptable. You can then try again. It takes really *nice* people to produce such a polite computer!

## Chapter Two

# Ins and Outs

A computer can be used as a calculator, but you didn't buy a computer to do the work of a calculator. One particularly important difference between the two lies in the way that they each produce a result. For example, on a calculator, pressing the keys to get  $2.4 \div 1.6 =$  will produce an answer. On the computer, it won't. For a start, there isn't a divide sign in the form of  $\div$ . A problem that is in the correct form for a calculator is not necessarily in the correct form for a computer. If you want to see anything placed on the screen of the computer (meaning the TV or monitor screen), then you normally have to *start* with the instruction word PRINT. The word has to be spelled correctly, though you can use the abbreviation ? which the Lynx and several other computers will accept in place of PRINT. We also have different signs for multiply and divide. The multiply sign is the asterisk \*, and the divide sign is the slash, /. You will find the \* on the : key (use SHIFT with :) and the / on the key that also carries the ? – you don't need SHIFT to get the /.

Now we can make some sense of calculations. Try:

PRINT 4.6\*2.2                      or                      4.6\*2.2  
(The Lynx does not need PRINT for a calculation.)

and then press RETURN. You can either type out the letters of PRINT or you can use the ? mark, but nothing happens until you press the RETURN key after typing the last '2'. The RETURN key is, as always, an over-to-you signal to the computer, meaning that you have done your part and the computer now has to act on it.

When we carry out actions like PRINT 8.7 / 1.4 or PRINT 476 + 223, then pressing the RETURN key will produce the result of these actions. This is a *direct command* – it is carried out directly when you press the RETURN key, and the only way you can repeat the action is by typing it all over again. We can also print letters on the screen – but only if what we want to print is enclosed by quotes. For example, PRINT"THIS IS A LYNX" will put these words on



the screen, under the instruction, when you press RETURN. Notice the form of the words, though. The quote marks are not printed, nor is the PRINT command. You will not, at this stage, get anything printed on the screen if you type PRINT X – but look out for more on this point.

Direct commands can be useful, but a *program* is very much more useful. A program is a set of instructions which the computer will carry out in order, one at a time. These instructions can be stored in the memory of the computer and then carried out for as long and for as many times as you like. A program is particularly useful if you want to perform a large number of different operations in a sequence, as you might want to do in calculating income tax deductions or arranging words in alphabetical order. A program is also useful if you want to carry out the same set of operations many times over. How, then, do we enter a program into the computer?

The computer is arranged to treat as a program instruction anything that starts as a number. We can use any number between 1 and numbers so large that they take a whole screen line to type! This is another unusual feature of the Lynx – most computers will not accept numbers for this purpose if they exceed 65536. Normally, however, we start at 100 and number in tens. This is because each instruction in the program uses its number, called the *line number*, as a guide to the order in which the instructions will be carried out. If we number in tens, as 100, 110, 120, 130 and so on, then second thoughts are easily catered for. If we need an extra instruction between lines 110 and 120, we can use line 115 – we could, in fact, use anything between 111 and 119. The Lynx goes further than most computers, in fact, by allowing you to use line numbers that contain decimal fractions. We can, for example, use 110.25 as a line number. This is *very* unusual, and it's not a good habit to acquire. On the other hand, if you simply have to have one more line ...!

- (a)        100 PRINT"THE LYNX"  
           120 PRINT"IS A REMARKABLE ANIMAL"  
           110 PRINT"COMPUTER"
- (b)        100 PRINT "THE LYNX"  
           110 PRINT "COMPUTER"  
           120 PRINT "IS A REMARKABLE ANIMAL!"

Fig. 2.1. Illustrating the PRINT action (a) as typed, (b) as rearranged by the computer.

Even if you don't type the lines in sequence, the computer will arrange them in sequence. Try typing the short program in Fig. 2.1,

exactly as it is shown. You will have to type in the line numbers, but we'll look at a way around that problem later. When you have typed each instruction line, press the RETURN key to signal to the computer that you want the line put into memory so that you can type the next one. At the end, type LIST and then press RETURN. You can abbreviate the typing of LIST (and you can abbreviate many other instructions) by typing ESC L, meaning press the ESC key and the L key at the same time. This brings up the word LIST on the screen and, by pressing RETURN, the program will list. Listing means that the lines of your program will be printed on the screen, in number order. This process is possible because the program is stored in memory, so that we can make a copy on the screen any time we want. If you find a mistake in the program at this point, take a look at Appendix C, which describes *editing*. Now we can see the effect of the program. Type RUN (or press ESC and M keys together), and then press the RETURN key. This carries out the instructions in sequence, producing three lines of text on the screen.

You'll find that if you repeat these steps several times then the screen fills with printing. This is because a new line is selected each time the instruction word PRINT is used. When the screen of the Lynx is filled with words, printing starts again at the top. This again, is unlike the action of any other computer. You can remove the clutter from the screen by typing CLS and pressing RETURN. The instruction CLS can also be used in a program. Add the line:

```
5 CLS
```

to the program of Fig. 2.1, and notice the difference it makes when it runs. An alternative to CLS is VDU4 – but that's longer!

## Prettier printing

You'll find that the Lynx prints rather slowly. The reasons for this are discussed later but you will notice an immediate improvement if you type TEXT and press RETURN before starting to program. If you are using a black and white receiver you should always use this procedure. Do not use this instruction if you are going to work with colour graphics (see Chapter 8).

The use of PRINT by itself is useful, but the result is far from neat. We can alter the appearance of printing on the screen by several additional instructions or marks that we call *print modifiers*. One of these print modifiers is the semicolon. Try the small program in Fig.

```

100 PRINT "THIS SHOWS";
110 PRINT " THE EFFECT";
120 PRINT " OF SEMICOLONS"

```

Fig. 2.2. The effect of semicolons to keep printing on one line.

2.2 – but start this time by typing NEW (then press RETURN) to remove the old program, then type AUTO (press RETURN). AUTO is a command that calls for automatic line numbering. It will make the first line of a program 100, and will number in tens from then on. You can escape from the sequence (unlike the Sorcerer's Apprentice!) by typing a RETURN with only the line number showing. When you type this program, pay particular attention to the spaces between the first quote marks and the first letter in lines 110 and 120. The effect of the semicolon placed after the final quote marks is to prevent printing from going to a new line. It will also remove spaces. If you add the line:

```
105 PRINT"CLEARLY"
```

with no space between the first quote mark and the letter C, you will find that the word CLEARLY is forced right up against SHOWS. We'll see this use of the semicolon illustrated further later on. Meanwhile, take a look at the program of Fig. 2.3. This illustrates the difference between a quantity placed between quotes and one which isn't. The quantity that we place between quotes is called a *string*, and it's always printed exactly as we type it. The quantity that is not between the quotes is, in this example, a piece of arithmetic which is worked out and the result printed. This happens because it is an operation on numbers, just as if we had typed PRINT 2\*3. The PRINT instruction affects everything that follows it, but in different ways.

```

100 PRINT "2*3 IS ";2*3
110 PRINT "150/5 IS ";150/5
120 PRINT "27.6-12.4 IS ";27.6-12.4

```

Fig. 2.3. Using quotes – whatever is enclosed in quotes will be printed exactly as it is.

Commas have a remarkable effect on printing, as Fig. 2.4 shows. A comma placed between two lots of items that are to be printed causes the printing to be done in columns. When you print numbers, this is straightforward, as line 100 of Fig. 2.4 shows. When you print strings, you must make certain that the commas are not included within the quotes. If they are, then they will be printed along with

## 18 Lynx Computing

```
100 PRINT 1,2,3,4,5
110 PRINT "LYNX","LYNX","LYNX","AND","LYNX"
120 PRINT 1,2,3,4,5,6
130 PRINT "TOO MUCH FOR","ONE","SET",
"OF","WORDS"
```

*Fig. 2.4.* The effect of commas to place items into columns.

other characters instead of causing the spacing effect. You can't get more than five columns, as line 120 shows. A sixth column lines up in the first column position, one line under. You can't get more than eight characters in a column either, as line 130 shows. It's likely that these numbers (5 columns, 8 characters) can be altered by some adjustments to the piece of program that causes these effects (the operating system), but this is definitely not for beginners!

### Tabbing and atting

PRINT can be modified even more noticeably by two more additions, TAB and @ (called *at*). TAB is short for TABulation, meaning the character position along a line. A Lynx line contains 40 positions of tabulation which are numbered 0 to 39. Position zero is on the left-hand side of the line, 39 is on the extreme right. We can use TAB to cause something to be printed so that its first character is at the TAB position. Take a look at Fig. 2.5, which uses TAB to

```
100 CLS
110 PRINT TAB 15;"LYNX LORE"
120 PRINT TAB 5;"TABULATION GIVES NEAT
ER PRINTING"
```

*Fig. 2.5.* The TABulation instruction.

place words so that they are centred on the screen line. Line 100 uses CLS to make certain that the screen starts blank, and the printing lines in 110 and 130 put the messages in place. TAB can be used anywhere in the line following PRINT, but the TAB number must be separated from the quantity that is being printed by a semicolon. You can use PRINT TAB 5; or PRINT TAB5; or PRINTTAB5; as you please, it makes no difference, but you can't omit the semicolon. That's computing!

How did I find the correct TAB number for these phrases? It's an old typist's trick (I meant a typist's old trick, madam), in which the characters (letters and spaces) are counted. The method is illustrated

in Fig. 2.6. Note while we are on the subject that you can have more than one TAB in a print line – Fig. 2.7 illustrates this point. When multiple TABs are used, you have to be careful that you are not trying to TAB backwards. For example, if you print a word of seven letters at TAB 6, then you can't expect to make the next TAB a TAB9. You don't need a semicolon between the final quotes of a word and the next TAB, but the computer won't reject the semicolon and it helps to separate the pieces of the PRINT line when you look at the listing.

- 
1. Count the number of letters and spaces in the phrase.
  2. Divide this number by two, take whole number only.
  3. Subtract it from 20.
  4. The result is your TAB number.

*Example:* The phrase "THIS IS A TITLE" has a total of 15 characters.  
 Divide this by 2 to get 7, ignoring the  $\frac{1}{2}$  left over.  
 Subtract from 20 to get 13, and so use TAB13.

---

*Fig. 2.6.* How a phrase is centred on a line.

```
100 CLS
110 PRINT "LYNX"; TAB 8;"ON"; TAB 13;
    "THE"; TAB 19;"FROWL"
```

*Fig. 2.7.* Using more than one TAB instruction after one PRINT.

You will notice in Fig. 2.10 that line 120 simply consists of the word PRINT with nothing following. The effect of this is to cause a blank line to be printed. You could, if you liked, space several lines down the screen in this way, but there is an easier way of carrying out this sort of action, the use of PRINT@. This is illustrated in Fig. 2.8, where PRINT@ is used to space the first

```
100 CLS
110 PRINT @ 54,50;"LYNX"
120 PRINT @ 0,90;"WALKING DOWN THE LINES"
```

*Fig. 2.8.* Using PRINT@. You have to remember to use the correct multipliers of 3 for the TAB value and 10 for the line (row) value.

heading in line number 5 from the top of the screen. PRINT@ doesn't, however, use the same columns as TAB or the normal line count of 25. To get the correct number for PRINT@, you have to multiply the TAB number that you would use by 3 and the line

number by 10. For example, if you wanted to print at TAB position 12 on line 4, then you have to use the numbers  $12 \times 3 = 36$  and  $4 \times 10 = 40$ , making the instruction `PRINT@36,40`. The column number comes first, then the row number, then a semicolon, and finally what you want to print. It doesn't matter if you leave a space between the @ and the first number (old TRS-80 users will heave a sigh of relief!), and you can have more than one @ in a line, as Fig. 2.9 shows.

```
100 CLS
110 PRINT @ 54,50;"LYNX"; @ 84,50;"CAPERS"
```

Fig. 2.9. More than one @ can be used in a line, as shown here.

You can therefore print what you want, where you want, on the screen. You must remember that if you want something printed exactly as you type it you will have to enclose it in quotes. If you want a piece of arithmetic worked out, you can place it following the `PRINT` with no quotes. The next thing that we have to look at now is how to get words and numbers into the computer.

## Wait for it!

Figure 2.10 is a program that prints your name on the screen. Now I don't know what your name is, but I can still write a program that will print it! The key to this achievement is the instruction `INPUT`, which is used in line 130. When the computer finds this instruction in a line, it stops and waits for you to type something. In this case, you are invited to type your name. You then have to signal to the

```
100 CLS
110 PRINT TAB 16;"MY NAME"
120 PRINT
130 INPUT "YOUR NAME, PLEASE";N$
140 CLS
150 PRINT N$;" -THIS IS YOUR LIFE!"
```

Fig. 2.10. Printing your name – but the name is coded.

computer that you have typed all you want to, and you do this, as you might expect, by pressing `RETURN`. The computer then continues on its way, and prints your name, with the famous phrase, in line 150.

Wait, though. What the computer is instructed to print in line 150 is `N$` and that doesn't exactly look like your name. This is an introduction to what we call a *variable name* – a short code that can

be used to represent a number or a name. The code name in this example is N\$, pronounced 'en-string', and this type of name is called a *string variable*. Line 130 *assigns*, meaning that the name which you type is coded as N\$. Wherever you have N\$ in the program after that, you will see your name printed. This remains stored even after the program has finished – try typing PRINT N\$, and then press RETURN. If you find that there are some letters missing from the end of your name – well, I'll explain that later.

Variable names are a very useful way of saving a lot of typing. Take a look at the example in Fig. 2.11. In line 110, we make the

```

100 CLS
110 LET S$="SMITH"
120 PRINT TAB 12;"THE FAMILY ";S$
130 PRINT
140 PRINT
150 PRINT TAB 5;"Donald ";S$
160 PRINT TAB 5;"Kirstie ";S$
170 PRINT TAB 5;"Gordon ";S$
180 PRINT TAB 5;"Sarah ";S$
190 PRINT
200 PRINT "Is your name ";S$;" ?"

```

Fig. 2.11. The S\$ family program. The use of a variable name saves a lot of typing!

variable name S\$ equal to (assigned to) the word SMITH. This has to be done in a different way, using the word LET, and enclosing SMITH in quotes. The INPUT method of assigning a variable does not need LET or the quotes. From line 120 on, each time S\$ is used, the word SMITH will be printed. When you've tried that, type another line:

```
115 LET S$ = "PLINGE"
```

and now run the program. Yes, PLINGE has replaced SMITH (happens all the time in the theatre), because line 115 comes later than line 110, and S\$ has been reassigned. That's why we call these quantities variables; we can vary what each one of them represents.

Now you may have had a problem in the program of Fig. 2.10. You won't have met the problem if your name is short, like TOM JONES, but if you typed BARTHOLOMEW PEREGRINE WINTERBOTTOM, then you will have found that only about half of your name was printed. This is because the Lynx is organised to chew strings in small mouthfuls. It can take 16 characters (remember that a space is also a character) but no more into a string variable. An intolerable limitation, Bartholomew? Well, you can get round it if you know how many characters you *might* want to use. Suppose

we want to allow up to 40 characters in the string N\$. We can instruct the Lynx to permit this by a line like:

```
105 DIM N$(40)
```

DIM means dimension, it's not a slur on your brainpower. This instruction causes the computer to reserve enough memory for 40 characters, which will be assigned to the variable name N\$ in Fig. 2.10. Lynx won't grumble if you use fewer than 40 characters, but if you attempt to use more than 40 the excess will simply be ignored. Since you can dimension a string to hold up to 127 characters, you don't have to feel restricted by this additional provision. Its great advantage is that it makes it much easier for the computer to work with string variables. You have to dimension each string name that you use, if you are likely to exceed your 'default' allocation of 16 characters, but you can dimension several in one line by using a DIM instruction such as

```
DIM A$(20),B$(40),C$(50)
```

Incidentally, if in Fig. 2.10 you typed MICKEY MOUSE in place of your name, the computer would obediently go through all the same motions. What was all that guff about computers taking over the world? Screwdrivers didn't take over the world, after all, though some people think that nuts did.

## Names and numbers

The Lynx offers you a choice of 26 variable names for string variables. These consist of all the upper-case letters (capitals) of the alphabet, and only one letter can be used for a variable name. We can also, as Fig. 2.12 illustrates, use a variable name for a number,

```
100 CLS
110 LET a=7
120 LET A=6
130 LET A$=" GREEN BOTTLES"
140 PRINT a;A$
150 PRINT "DROP ONE AND YOU HAVE ";A
```

*Fig. 2.12. Using different variable names, A, a, and A\$.*

with a greater choice of letters, both the upper-case and the lower-case letters. The letters a and A that have been used as number variables are not confused with the name A\$ which is used as a strong variable – the \$ sign sees to that. This is the explanation, incidentally, of why we could use PRINT 5 and get the number 5 on



the screen, but `PRINT X` did not place `X` on the screen. The computer treats `X` always as a number variable. Unless you have assigned some value to `X`, then as far as the computer is concerned it doesn't exist, and you get the 'unassigned variable' message. Unlike many other computers, the Lynx does *not* allow you to use longer variable names. Any attempt to use a variable name of more than one letter will be greeted with a 'syntax error' message.

While we're on the subject of variables, we can take a closer look at this `INPUT` instruction. We saw in the course of Fig. 2.10 that `INPUT` would cause the computer to wait for you to type something, and then would hold on, requiring you to press `RETURN` before going on with the program. There is another effect. When the program waits because of an `INPUT` instruction, a question mark is printed on the screen as a reminder (a 'prompt') that you have to do something. This action is automatic, and you should try to make the question mark look as if it were part of the phrase that is being printed. Figure 2.13 illustrates this – the first

```
100 CLS
110 PRINT "What is your surname"
120 INPUT S$
130 PRINT "What is your first name";
140 INPUT F$
150 PRINT F$;" ";S$
160 PRINT "Any relation to Bartholomew "
    ;S$;"?"
```

Fig. 2.13. Using `INPUT` to phrase questions more neatly.

`INPUT` places a question mark on the line below the question, which looks clumsy. The second question is phrased so that the question mark appears on the same line. This has been done by using a semicolon following the `PRINT` instruction in line 130. The alternative method is, as used in Fig. 2.10, to make the `INPUT` instruction carry out the printing. When this is done, the semicolon has to be used to separate the printed text from the variable name, as for example:

```
INPUT"Your name is ";N$      (watch the space!)
```

A single `INPUT` can be used to provide more than one answer, as Fig. 2.14 illustrates. You are asked to type two numbers, which will then be multiplied. In line 140, the `INPUT` instruction prints the reminder 'Your numbers are–', and then you can enter *two* numbers. You can do this in two ways. One way is to type the first number, then a comma, then the second number and then press `RETURN`. The second way is to type the first number, press

```

100 CLS
110 PRINT TAB 10;"MULTIPLYING NUMBERS"
120 PRINT
130 PRINT "Type two numbers please"
140 INPUT "Your numbers are-";A,B
150 PRINT
160 PRINT "The product is ";A*B

```

*Fig. 2.14.* An INPUT that expects two replies.

RETURN, and then type the second number and press RETURN. When you use the second method, the Lynx will print another question mark after you have entered the first number as a reminder that there is more to come. Notice the form in which the variable names A and B have to be written in line 140. They are separated by a comma. This means that you have to be careful to avoid using commas when you reply to an INPUT. If you are asked to enter your name, for example, by a line that reads:

```
INPUT"Your name is- ";N$
```

then you can't expect to type SINCLAIR, IAN R (press RETURN) and have all of the name accepted. Only the word before the comma will be accepted, because the computer is designed to take what follows the comma as being assigned to another variable – and there isn't one! The computer will not remind you in any way of this (some print the message 'excess ignored'), so it's advisable to carry out some sort of check by printing the value of a variable after it has been entered by the use of INPUT.

There's another error that can be caused in an INPUT. Suppose you have in your program the line:

```
INPUT n
```

which expects you to enter a number. When you press RETURN, the computer will check this entry. The variable n is a number variable, and only a number can be assigned to a number variable. If you type a letter, you will get the error message 'undefined variable', and if you type several letters you will get 'syntax error'. In either case, the computer doesn't give up – the question mark is printed again, and it waits for you to type the correct item – a number. The behaviour of the Lynx in this situation is most commendable – many computers simply stop the program when they encounter a small problem like this! You can, however, quite cheerfully enter a number when a string variable is used in the INPUT instruction. If, for example, your program contains INPUT A\$, then any characters that you like to type will be acceptable, letters, numbers, spaces – but watch out for commas!

## Loose ends

Three loose ends need to be tied up at this point. One is that you must type only one instruction per line number. Many computers allow you to use more than one instruction per line, with a colon (:) used to separate them. This can lead to lines of instructions that are very hard to read, and Lynx keeps strictly to one instruction in each line.

The second point is that you can use LET to carry out more than one assignment. You can type, for example:

```
LET a = 4, b = 7 , C$ = "NAME"
```

This is something that other computers do not permit.

The final point is that you don't have to type everything in upper-case letters. When the Lynx is switched on, the letter keys give upper-case letters only. If you tap the SHIFT LOCK key once, however, you will see that the letter keys then give lower-case letters. If you type an instruction word like 'print' in lower-case, then you will find that the Lynx has converted it to upper-case when you list the program. This avoids one of the shortcomings of other computers where if you type an instruction in lower-case it is treated as a syntax error.

## Chapter Three

# Counting, Listing and Looping

A lot of people associate computers with numbers and with numbers alone. The fact is that computers are used a lot more for string operations, involving names, than just for calculations. Nevertheless, the ability to make use of numbers is the foundation on which computing is built, and this chapter is concerned with some of the operations that make use of numbers, though not always in the mathematical sense that you might expect.

Top of our list has to be counting. As the Channel Ferry passenger said of his sandwiches: 'You count 'em all in, and you count 'em all out again'. One simple provision that is made for counting by computers is altering a variable value. If you type the instruction:

```
LET z = z + 1
```

then the effect is to add 1 to the value that variable had, and to call the result by the same name, z. If z was 10, for example, then `LET z = z + 1` would cause the value of z to become 11. It looks weird in the form `LET z = z + 1`, but if you think of the `=` as meaning 'becomes' when the same variable name is used on each side of the `=` sign, then you will soon become used to it. We could also write:

```
LET x = x - 1
```

which causes the value of variable x to be reduced by 1. We can also have instructions like:

```
LET y = y*2
```

or

```
LET w = w/3
```

to change the values of these variables.

Now counting is a repetitive process, and all computers provide in some way for repeating processes. The simplest way that is used in

BASIC, and practically all other computer languages, is by the use of the instruction GOTO. GOTO means what it says – go to a different line. Normally when the computer has carried out an instruction, it will go to the next line number in ascending order. GOTO cancels this automatic process for one step, and forces the computer to go to a line whose number follows the GOTO instruction. Take a look at the program in Fig. 3.1 to illustrate this. It is a program that keeps a running total for you, adding numbers as you type them in and press RETURN. It uses the variable T (for total), which has to be set in line 110 equal to zero, and is added to each time a number is entered. As it is written here, it will go on for ever and, unless you know the trick, it is very difficult to stop. When a program seems to be unstoppable, pressing the ESC key will usually stop it, but special arrangements have to be used to stop a program that contains an INPUT. This is deliberately done so that an inexperienced user will not stop a program by mistake – it's no fun if you have just entered one hundred sets of tax codes and incomes, and the other guy comes and says: 'It's stopped, with a funny message, so I switched off ...'. The way out of the endless program that contains an INPUT is to type a number, press the ESC key, and then press RETURN while you are holding the ESC key down. If you type a letter by mistake, the computer will give the message 'undefined variable', and give you another chance. If, however, you find that the ESC and RETURN procedure doesn't get you out of the jam, then you will have to switch off – the key marked BREAK doesn't do anything (on the present Lynx, anyhow).

```

100 PRINT TAB 17;"TOTALS"
110 LET T=0
120 PRINT
130 PRINT "Type a number, then press
RETURN"
140 PRINT "The program will keep the
total."
150 INPUT N
160 LET T=T+N
170 CLS
180 PRINT "The total so far is ";T
190 GOTO 120

```

*Fig. 3.1. A simple but unsatisfactory running-total program.*

Having to take measures like these to stop a program is a pretty poor state of affairs, and we ought to be able to arrange our program better. One way that all computers provide for is to test what you have entered. If what you entered is an acceptable number, then the

program continues. If it isn't acceptable, the program stops. The problem now is, how do we instruct the computer to treat an entry as unacceptable?

The answer to that one lies in the use of a test, consisting of the instruction words IF and THEN. The IF part of this has to be followed by some condition, like:

IF x = 0      or      IF A\$ = "END"

and the THEN part has to be followed by what you want the computer to do when the IF test succeeds. Figure 3.2 shows a version

```

100 CLS
110 PRINT TAB 17;"TOTALS"
120 LET T=0
130 PRINT
140 PRINT "PLEASE ENTER A NUMBER";
150 INPUT N
160 IF N=0 THEN END
170 LET T=T+N
180 CLS
190 PRINT "Total so far is ";T
200 GOTO 130

```

*Fig. 3.2.* Using an IF test to improve the totalling program.

of the totalling program which allows us to end it by entering a 0 instead of any other number. The test is made immediately after the INPUT in line 150, so that what you have entered and assigned to the variable is tested in line 160. If this quantity is zero, the program ends. We could have caused a completely different action here by using a GOTO following THEN. We could, for example, have:

IF n = 0 THEN GOTO 210

and then written a line 210 which dealt with this. Figure 3.3 shows

```

100 CLS
110 PRINT TAB 9;"MORE TOTALS"
120 LET T=0
130 PRINT
140 PRINT "Please enter a number"
150 PRINT "(0 will end the program)"
160 INPUT "Your number ";N
170 IF N=0 THEN GOTO 230
180 LET T=T+N
190 CLS
200 PRINT "Total so far is ";T
210 GOTO 130
220 END
230 PRINT "End of program - total is ";T

```

*Fig. 3.3.* A further improvement using a separate print line.

the program modified to use this idea. Note the use of the line 220 END. This ensures that line 230 can never be carried out by accident as the computer goes on its way. In this particular example, of course, the line 210 would ensure that the computer could not carry out any line number beyond 210 except as a result of a GOTO instruction. In other programs, however, it might be possible for the computer to carry out the last line even if it had not been instructed to go there by a GOTO. Using END just ahead of lines like 230 prevents this type of mistake, called 'crashing through'.

The idea of counting is so important that all computers use special instructions for this purpose. The instruction words for one very important counting action are FOR and NEXT, and the principle is to set up a number variable to carry out a count. Another important point, though, is that all of the lines between the FOR part and the NEXT part will be carried out each time the number count is changed.

Take a look at Fig. 3.4. This uses FOR...NEXT in two ways. The

```

100 CLS
110 PRINT TAB 16;"COUNT-UP"
120 PRINT
130 FOR N=1 TO 10
140   PRINT @ 60,100;N
150   FOR J=1 TO 1000
160     NEXT J
170 NEXT N
180 PRINT "Done!"

```

Fig. 3.4. The FOR...NEXT loop instructions.

main loop, as it is called, starts in line 130, with FOR N = 1 TO 10. This means that we are using a variable name N to carry out a count from 1 to 10 in steps of 1 (we don't have to specify a step of 1). We must ensure that while this count is being carried out, we don't do anything that will change the value of N, though we can print this value. We carry out the printing in the same part of the screen each time, specified by the use of PRINT@ in line 140 – remember the formula of three times TAB number and ten times row number.

Lines 150 and 160 make a different use of the FOR...NEXT loop, however. The instruction simply calls for a count from 1 to 1000, using a different number variable, J. Nothing is done between the FOR J = 1 TO 1000 in line 150 and the NEXT J that follows it in line 160, so all that these lines do is to create a time delay. Counting, like all computing processes, takes time, and by using a count to 1000 we can get a delay of about one second. The NEXT J in line 160 is the NEXT for the loop that started in line 150. NEXT must always

apply to the loop that started most recently. The NEXT N in line 170 is for the loop that started in line 130. You must label the NEXT instructions with the variable names, because Lynx will reject the instruction if you do not. The only snag is that if you get the labels reversed, the loops can't operate correctly, and the Lynx will give you a NEXT without FOR message – what it means is that you have the wrong NEXT for the FOR! Placing one loop inside another is called *nesting*, and the Lynx's nesting follows an inflexible rule. The rule is that each loop has to be totally enclosed by another when you have more than one loop – you simply can't have:

```
FOR N = 1 TO 5
FOR J = 2 TO 6
(other instructions)
NEXT N
NEXT J
```

because the loop that uses J is not enclosed in the loop that uses N, and the loop that uses N is not enclosed in the loop that uses J. This is cuckoo nesting, and it isn't allowed. Note, incidentally, while we're on the subject of loops and nesting, how the Lynx lists these lines. The instructions within a loop are indented (TABbed to the right) in their lines to make it obvious that these are the repeated lines. If one loop is within another, the inner loop instructions are indented more than the outer ones. This action is carried out when any of the Lynx's many types of loops is listed.

While we're counting, take a look at Fig. 3.5, which brings a new

```
100 CLS
110 PRINT TAB 15;"COUNTDOWN"
120 PRINT
130 FOR N=10 TO 1 STEP -1
140   PRINT @ 60,100;N
145   PAUSE 10000
150   PRINT @ 60,100;"  "
160 NEXT N
170 PRINT "Blast-off!!"
180 PRINT
190 PRINT "(N is now ";N;" )"
```

Fig. 3.5. A countdown program which leaves variable N equal to zero.

direction to counting. This is a countdown program, and it introduces a new instruction, STEP. Normally when you make use of the FOR...NEXT loop, the count is upwards in steps of 1, but by using STEP followed by a number, you can make the step any value you like, positive, negative or fractional. In Fig. 3.5, we have made the count start at 10, and go down to 0 in steps of -1, so ensuring a



countdown. Just to stop you getting too confident, there's another novelty in line 145 in the shape of PAUSE. For a Lynx, I think that PAWS might have been more appropriate, but PAUSE is the word you have to use. It's followed by the number 100000, which causes a delay of about one second. The delay is proportional to the number, so that if you use 50000, you get  $\frac{1}{2}$  second, if you use 1000000 you get 10 seconds, and so on. It saves having to use another number variable in a FOR...NEXT loop.

The countdown proceeds, until 0 is reached, and the words "Blast-off!!" are printed. What follows is interesting, though, because you sometimes want to use the value of N after a loop stops. In this example, the value of N is 0, which shows that the loop finished when the count reached zero. This does *not* apply to a count-up loop, nor does it apply to a countdown which finishes at any number other than 0. If you use FOR N = 1 TO 100, then the value of N after the loop has ended will be 101. If you use FOR N = 50 TO 5 STEP -1, then the final value of N will be 4. The value is always one step on when the loop finishes, except when the loop finishes at 0. This type of behaviour can sometimes cause problems unless you are aware of what happens.

An illustration will, as usual, help to make things clearer, and we can introduce a new instruction. Figure 3.6 shows a program that

```

100 CLS
110 FOR N=1 TO 10
120   READ G$
130   PRINT G$
140 NEXT N
150 PRINT "A total of ";N-1;" items."
160 DATA Potatoes,Carrots,Cabbage,Bruss
    els,Cauliflower
170 DATA Onion,Celery,Tomatoes,Cucumber,
    Lettuce

```

*Fig. 3.6.* A simple shopping-list program. Notice how the value of N is one more than the actual total.

will read words out from a list and display them on the screen. As it is shown here, it might be part of a shopping list, but you could make it anything you wanted. The counting loop starts in line 110 with FOR N = 1 TO 10, so that we are working with a count of ten. In line 120, we have the new instruction, READ. This causes the computer to look for another instruction word DATA at the start of a line. DATA will be the word at the start of a list of items that are separated by commas, and the computer keeps track of this list. The first time that READ is used, the computer looks for the first item in the lowest numbered line that starts with DATA. In this example,

it's line 160, and the item that is read is Potatoes. The DATA can consist of strings or numbers, and strings don't need quotes around them when they are put into a DATA list like this. Once again, the comma is used to separate items in the list, so that you can't have items that contain commas!

Another point is that you must read a string into a string variable. If you have READ n, then you can't have DATA Lynx. This sort of thing is called a 'type mismatch' but on my Lynx it caused a syntax error in the line that contained the READ. This is rather unusual.

Back to the program. As each item is read, it is printed by line 130, and the NEXT in line 140 causes the loop to be repeated until N exceeds 10. Each READ will be of a different item, because once the computer has read one item, it will automatically choose the next one for the next READ. This happens even if the next item is on another DATA line, as the example shows.

Now try a variation. Type the new line:

```
105 RESTORE 170
```

and then RUN the program. It makes a considerable difference, doesn't it? The program ignores line 160, because the instruction RESTORE 170 has caused it to start the DATA 'pointer' at this line instead of at 160. It won't go to 160 from 170, so it's caused another problem too. The program ends with an error message. 'Out of data' means that you specified that you wanted to read more items than were in the DATA list, so the computer can't carry out the READ instruction.

The ability to RESTORE to a particular line number can be very useful. Take a look at the example in Fig. 3.7 which, as always, introduces another new idea. This is a longer program than we have used before, and you might want to correct typing mistakes after you list it. Editing, which is the term meaning adding, deleting or changing characters in a line, is dealt with in Appendix C. If you see a mistake as you are typing a line, however, you can simply use the DEL key to backspace and delete the mistake. You can also skip over characters you don't want to delete by using the arrow keys.

The program starts with a title and brief instructions. It will print a list of days of the week in three different languages: English, French or Spanish. The choice has to be made by typing a single letter – E, F or S.

The new instruction occurs in line 180. GET\$ is an instruction that causes the computer to scan its keyboard, looking for a key being pressed. If you don't press a key, then it continues to wait, but

```

100 CLS
110 PRINT TAB 12;"DAYS OF THE WEEK"
120 PRINT
130 PRINT "Would you like -"
140 PRINT TAB 10;"English,"
150 PRINT TAB 10;"French,"
160 PRINT TAB 10;"or Spanish ?"
170 PRINT "Please choose E,F or S"
180 LET A$=GET$
190 IF A$="E" THEN RESTORE 300
200 IF A$="F" THEN RESTORE 320
210 IF A$="S" THEN RESTORE 340
220 IF NOT A$="F" AND NOT A$="E" AND NOT
A$="S" THEN GOTO 280
230 FOR N=1 TO 7
240 READ D$
250 PRINT N;" ";D$
260 NEXT N
270 END
280 PRINT "Incorrect selection -try again"
290 GOTO 170
300 DATA Monday,Tuesday,Wednesday,Thurs
day,Friday,Saturday,Sunday
320 DATA Lundi,Mardi,Mercredi,Jeudi,Ven
dredi,Samedi,Dimanche
340 DATA Lunes,Martes,Miercoles,Jueves,
Viernes,Sabado,Domingo

```

*Fig. 3.7.* A day-of-the-week program, giving you the choice of three languages.

when you *do* press a key the waiting is over. You don't need to press RETURN when you use this instruction in place of INPUT, and that makes it ideal for fast one-key replies. Whatever key you have pressed at this stage has its character assigned to A\$, so that if you pressed the 'E' key, then A\$ has the value 'E' just as surely as if you had typed LET A\$ = "E" or had pressed the E key, then RETURN at an INPUT step.

Lines 190 to 220 then test your answer. If the answer is E then RESTORE 300 selects the data lines that contain the start of the English list. If you selected F, then RESTORE 320 starts any reading of DATA with the French list; similarly by typing S you select the Spanish list. Line 220 deals with the problem of mistakes – what if the key that you pressed was none of these? This is done by testing again. If the answer was NOT E or F or S, then line 280 is carried out. Note the unusual form that the Lynx uses for this test, IF NOT A\$ rather than IF A\$<> which is used by other computers.

If your selection was reasonable, though, the correct DATA line is selected, and the loop in lines 230 to 260 prints a list of the numbers and names of the days, making MONDAY day 1. Take a good long

look at this program, and how it works, because it contains a lot of useful methods that you can use in your own programs.

### Other loops

The FOR...NEXT loop is a useful one, but it is controlled by a number count, and we don't always want this. We might, for example, want to have a loop that would read DATA from a list until some 'dummy' item like  $\emptyset$  or 999 or "X" was read. A program like the example in Fig. 3.8 would do this. The loop is tested by line

```

100 CLS
110 READ A$
120 PRINT A$
130 IF A$="X" THEN END
140 GOTO 110
150 DATA CHABLIS, SAUTERNE, PIESPORTER
160 DATA NIERSTEINER, LIEBFRAUMILCH
170 DATA HAUT-POITEAU, MUSCATEL
180 DATA X

```

*Fig. 3.8.* Using a 'terminator' item, X in this case, to stop a loop.

130, which will end the program when the item "X" is read. The loop is constructed by the use of GOTO, and if we wanted to continue the program (to do something else) after printing the names of the goodies, we would have to use a different line 130, such as:

```
IF A$ = "X" THEN GOTO 200
```

with a continuation starting in line 200.

Now the trouble about GOTO is that it makes programs very difficult for humans to follow, though the computer is perfectly at home with its use. The difficulty is that GOTOs make it almost impossible to read down the lines of a program and see what it does. When you come to a line that contains a GOTO you have to skip over lines to find where the GOTO goes, and you then have to find out how to get back to the rest of the program. Some of the programs that were written in the early days (1977-78) of home computing were so full of GOTOs that disentangling them needed the combined skills of a snake-charmer and a spaghetti-knitter. Several modern computers have introduced into BASIC the types of loops that are used in other programming languages. These types of loops ensure that you very seldom need to use GOTO. One of these loops is the REPEAT...UNTIL loop.

Take a look at Fig. 3.9. In line 110, the instruction REPEAT

```

100 CLS
110 REPEAT
120   READ A$
130   PRINT A$
140   UNTIL A$="X"
150 DATA HAIG,JOHNNY WALKER,CUTTY SARK
160 DATA TEACHERS,BELLS,FAMOUS GROUSE
170 DATA X

```

*Fig. 3.9.* Using a REPEAT...UNTIL loop. This is much more satisfactory than the use of GOTO, but a few machines have this instruction.

means that every line between this instruction and the line that contains UNTIL will be repeated – until a condition is satisfied. The condition is in line 140 – UNTIL A\$ = "X". When the letter X is found, the repeating stops, and the loop finishes, so that the computer carries out the line that follows UNTIL. Snags? Yes, there is one in the program as it stands – the letter X gets printed as if it were one of the items. That's only because the test UNTIL A\$ = "X" does not take place until line 140, and A\$ has been printed by that time. If we were not printing it directly, that wouldn't matter but, in this example, it makes the list look rather amateurish. We can, however, easily sort it out by reversing the instructions in lines 120 and 130.

The REPEAT...UNTIL loop is a useful one, but the Lynx can purr with satisfaction because it alone (so far) among BASIC programmable computers has another type of loop. It's a slightly different type of loop in which the test is made before anything is repeated. The instruction words for this loop are WHILE and WEND.

The test word is WHILE, which has to be followed by a condition. Every instruction that lies between WHILE and the word WEND (WHILE-END) is repeated until the condition fails. Figure 3.10 shows this applied to our list type of program. Since A\$ has not been assigned when the program starts, it can't have the value of "X". The test in line 110 is the Lynx way of phrasing the test WHILE A\$ IS NOT "X", and while this is true, A\$ is printed and then a new value

```

100 CLS
110 WHILE NOT A$="X"
120   PRINT A$
130   READ A$
140 WEND
150 DATA GLENFIDDICH,GLENMORANGIE
160 DATA LAPHRUAIG,ISLAY MIST,GLENDUFF
170 DATA X

```

*Fig. 3.10.* A WHILE...WEND loop in action. The test is made in the first line of the loop.

read from the DATA list. The WEND instruction sends the program back for another loop providing A\$ is not equal to "X", and this looping will continue until the READ A\$ in line 130 does read "X". When this happens, it is detected in line 130 after WEND has returned the program to this line, and the program then jumps to the line following WEND – in this example, it ends.

```

100 CLS
110 PRINT"Will you want to repeat this
?"
120 PRINT"Type Y or N"
130 LET A$=GET$
140 REPEAT
150   PRINT"This is a repeat!"
160 UNTIL A$="N"
170 PRINT"END"
180 PRINT" Now try again..."
190 PRINT"Type Y or N"
200 LET A$=GET$
210 WHILE NOT A$="N"
220   PRINT"This is a repeat"
230 WEND
240 PRINT"END"

```

*Fig. 3.11.* Showing the difference between REPEAT...UNTIL and WHILE...WEND loops. You will have to use ESC to get out of the loop if you type Y!

Now it's sometimes an advantage to test at the end of a loop and it's sometimes an advantage to test at the beginning. One point about this is that when a loop is tested at the end, as the REPEAT...UNTIL loop is, all the instructions of the loop will be carried out at least once, whether you want them or not. Consider the example in Fig. 3.11. Lines 110 and 120 ask you to type Y or N in reply to a question about repeating. Now when the REPEAT...UNTIL loop in lines 140 to 160 is used, then whether you answered Y or N, the phrase 'THIS IS A REPEAT' will be printed. If you typed Y, you will then have to use ESC to stop the loop, because there's no way of altering A\$. When the program comes to the WHILE...WEND loop in its second part, lines 180 onwards, the phrase will be printed only if you type Y, not if you type N. This illustrates the difference between testing at the start and testing at the end of a loop.

Finally, we've looked at conditions rather too briefly. Figure 3.12 lists the conditions that can be applied for comparing numbers and strings. The conditions that can be used for comparing numbers are very much more numerous in the Lynx than the string comparisons, but the few string comparisons allow the essential comparisons to be made. It's easy to see what = and NOT mean; two strings are equal if

*Comparing numbers:*

=	identical to	
<	less than	
>	greater than	
<=	less than or equal to	} signs must be in this order
>=	greater than or equal to	
<>	not equal	
NOT	not equal	
AND	more than one condition	
OR	more than one possibility	

*Comparing strings:*

=	identical to
>	greater than
NOT	negative result

*Fig. 3.12.* The comparison conditions for testing.

their characters are identical, and not equal otherwise. It's less easy to see what the greater-than sign,  $>$ , means. The answer is that it signifies the position of a character in alphabetical order. Each letter of the alphabet is allocated a code number, called its ASCII code, and it's these code numbers that are stored, not letters. The ASCII (pronounced ASKEY) codes that are used by the Lynx are listed in the manual, and also in Appendix B. In the ASCII code, the letter A is coded as 65, and B as 66. By this reckoning, then, B is greater than A. When strings of more than one character are compared to find if one string is greater than the other, the codes for the first letters of each string are compared. If one is greater than the other, the comparison ends. If the first letters are identical, then the second letters are compared, and so on until a difference is found (or not, as

```

100 CLS
110 PRINT "STRINGS"
120 LET X$="QWERTY"
130 PRINT "TYPE A WORD, PLEASE"
140 INPUT Y$
150 IF Y$=X$ THEN GOTO 200
160 IF Y$>X$ THEN PRINT Y$;" comes lat
er in alphabetical order."
170 ELSE PRINT Y$;" comes earlier in a
lphabetical order."
180 END
200 PRINT "They are identical!"
210 END

```

*Fig. 3.13.* Comparing strings to find the correct alphabetical order.

the case may be). Figure 3.13 illustrates this comparison. It allows you to type a word, and have its position assessed in comparison to a nonsense word that is held as X\$.

Comparison of numbers is easier, and there is also a very useful instruction SWAP, which will exchange the values of two number variables. Figure 3.14 shows this in action, sorting two numbers into order. Line 130 tests first for equality, and then line 140 checks to find if A is greater than B. If it is, they are interchanged so that line 150 will give the correct order no matter what numbers you used.

```

100 CLS
110 INPUT "Please type a number ";A
120 INPUT "...and another,please ";B
130 IF A=B THEN GOTO 200
140 IF A>B THEN SWAP A,B
150 PRINT "Correct order is ";A;" then
    ";B
160 END
200 PRINT "They are EQUAL!"

```

*Fig. 3.14.* Putting numbers into the correct order.



## Chapter Four

# Numbers and More Numbers

Even in programs that appear to be concerned only with producing patterns on the screen, the use of the ‘number functions’ of the Lynx is important, and that’s the starting point for this chapter. A number function as far as we are concerned, is an operation that produces a number as its result – multiplication is a simple example. When you multiply two numbers together, you get another number, called the product, as a result of the operation; similarly when you add two numbers you get another number, the sum.

The first of the other number functions that we’ll look at is one that generates a *random number*. A random number means one which is picked in such a way that what you get is decided by luck, pure chance. A computer is a machine that runs to strict rules, so that a genuinely random number, one that would satisfy the conditions laid down by mathematicians, is almost impossible to obtain by purely computer methods. We can, however, get numbers which are near enough to being random for most of our purposes by using the instruction word RAND. Wherever RAND occurs in a program, followed by a number in brackets, the machine will allocate a whole number which lies in value between 0 and the number in brackets, less 1. Suppose, for example, that you used:

```
LET a = RAND(10)
```

The value of a that you find when this instruction has been executed can be anything from 0 to 9 (1 less than 10). If you want a number that lies between 1 and 10, then `LET a = 1 + RAND(10)` will provide it – which is the reason for the 1-less-than-10 business. Note also the reference to the instruction RANDOM in Chapter 10.

Now take a look at Fig. 4.1 which shows RAND being used to select heads or tails. The random part is in line 140, where `LET a = 1 + RAND(2)` will select a number which is either 1 or 2. We could just as easily have used `RAND(2)` by itself, giving 0 or 1, but I wanted to

```

100 CLS
110 PRINT TAB 13;"HEADS OR TAILS"
120 PRINT
130 REPEAT
140   LET a=1+RAND(2)
150   IF a=1 THEN PRINT "HEADS"
160   ELSE PRINT "TAILS"
170   PRINT "Again? Type Y to repeat."
180   UNTIL NOT GET$="Y"
200 PRINT "They are EQUAL!"

```

Fig. 4.1. Using RAND to select heads or tails.

illustrate the principle. Line 150 ensures that HEADS is printed if  $a = 1$ . Line 160 ensures that TAILS is printed if there is any other value – since the only other possible value is 2, 2 means TAILS. The whole program is part of a REPEAT...UNTIL loop, and the test is for the value of GET\$ obtained in line 180. If you type anything but 'Y', then the loop stops.

RAND is very useful if you want whole numbers, but every now and again a program needs random numbers that are fractional or which contain fractions. The Lynx has another random number function, RND, which we can use for this purpose. RND gives a number between 0 and 1, always less than 1 and with seven places of decimals displayed.

Take a look at Fig. 4.2. This simulates the results of measuring ten

```

100 CLS
110 PRINT TAB 9;"RESULTS OF INSPECTION"
120 PRINT
130 FOR N=1 TO 10
140   LET a=RND
150   LET b=60+a
160   PRINT "Item ";N;" measured ";b;"
      mm."
170 NEXT N

```

Fig. 4.2. Simulating the tolerances on cut lengths – a useful educational program!

samples of wood strip cut to a nominal 60.5 mm. In line 140, LET  $a = \text{RND}$  means that  $a$  will have some value less than 1, picked at random. In line 150 this quantity  $a$  is added to 60 to give a quantity that can range from 60 to almost 61. This should give a range of values that is centred around 60.5. The average will be 60.5 if the numbers are truly random, but this will be true *only* if the numbers are truly random *and* if we pick a large number of samples. How about a program to find the average value of 1000 of these numbers?

Dealing with RND brings us to another useful function, called

INT. INT means 'integer part of', meaning that anything following the decimal point is ignored, and only a whole number taken. The number that you want INT to work on is enclosed in brackets following INT. For example, INT(27.3) is 27, and INT(-3.7) is -3. One of the main uses for INT is avoiding an excessive number of decimal places. Suppose, for example, that you have to calculate the effect of adding VAT to a bill. The bill comes to £49.28, and 15% of this is £7.392, making the total £56.672. When we do this calculation using a calculator or by looking up tables, we knock off the odd .002, and make the bill £56.67 (or £56.68 if we feel that way inclined), but the computer will not do this unless we instruct it to do so.

```

100 CLS
110 PRINT TAB 15;"ADDING V.A.T."
120 PRINT
130 INPUT "Please type a money quant
y";B
140 LET X=B+15*B/100
150 PRINT"This comes to ";CHR$(96);X
160 LET Y=(INT(100*X))/100
170 PRINT"We make it ";CHR$(96);Y

```

Fig. 4.3. Rounding off a money quantity – essential if you deal with VAT.

Figure 4.3 shows this process in action. You type in a money quantity, but not the £ sign, in line 130, line 140 adds VAT at 15%, and then line 150 shows what this would be. Line 160 then removes the fractions of a penny. The action operates like this. Suppose we have a number such as 14.726. One hundred times this quantity is 1472.6, and the INT of this is 1472. One hundredth of this is 14.72, which is the original quantity with the .006 chopped off. If we want to round up rather than just chop, then line 160 can be changed to read:

$$\text{LET } Y = (\text{INT}(100 * X + .5)) / 100$$

Why? Well, 14.726 would be multiplied by 100 to get 1472.6, and .5 added to this gives 1473.1. The INT of this is 1473, and 1/100 of this is 14.73. A quantity like 14.724 would still be rounded to 14.72 – try it! Next mystery – how did the £ sign appear on the screen? The answer is that the £ sign has the ASCII code number of 96 in the Lynx, and the instruction which produced it, CHR\$, will be dealt with later. Keep it in mind when you write these finance programs.

The quantities that we have been working out in these examples bring up another point which affects all computers and calculators – precedence. Suppose we have an expression, meaning a whole set of

arithmetical actions. Typically, we might have something like:

$$y = 32 + 24 \times 3.6 - 4 / 1.5 + 2$$

Now what we get as a result of this depends on how we attack it. If we simply take a left-to-right order, we will start by adding 32 and 24 to get 56, then multiplying by 3.6 to get 201.6, then subtracting 4, giving 197.6, dividing by 1.5 to get 131.733, and then adding 2, resulting in 133.733. This is *not* the method that we ever use, however. My calculator, doing these operations in left to right order, gives 117.3. Why? It has been designed so that it does operations like multiplying and dividing *before* it adds or subtracts. When you press the keys for  $32 + 24$ , it does not add when you press the '×' key. Instead, it works out the  $24 \times 3.6$ , which is 86.4, and then adds in 32 to this to get 118.4. Similarly, it works out  $4 / 1.5$  before subtracting this quantity.

Computers are also designed to follow this order of precedence. The most important operation is raising a number to a power (called *exponentiation*) like  $3^4$  or  $1.6^3$ . This is programmed in the Lynx by using two asterisks, such as  $3^{**}4$  or  $1.6^{**}3$ , for example. When this step is part of an expression, it will be carried out first. Next in order comes the action of making a quantity negative, so that a -6 will get its sign attached at this stage. Next in order of importance of the ordinary arithmetic actions are multiplication and division, and then come addition and subtraction. The comparisons then follow. The whole set is illustrated in the Manual on page 89 if you want to refer to them.

If all the operations in an expression have the same importance (as they would if they were all additions and subtractions, for example), then they are performed in a left-to-right order. We can, however, force the computer to carry out actions in a different order by enclosing part of an expression in brackets. Whatever is enclosed in brackets is done *before anything* else, and if we use one set of brackets within another, the *innermost* set of brackets has priority. For example,  $24 \times (1.2 - 14 / (3 - 1.4))$  will be carried out by taking  $3 - 1.4$  (the innermost brackets), giving 1.6, then dividing this into 14 to get 8.75 and then subtracting from 1.2 to give -7.55. This completes the bracketed actions, and then multiplying by 24 gives -181.2, the final answer. Take a close look at the way that we used these brackets in the examples that involved the use of INT.

Having got that out of the way, it's time to look at some more functions. ABS produces an absolute value, meaning that there will be no negative sign.  $\text{ABS}(-1.7)$  is 1.7, for example, a feature that can be useful if you need to ignore or replace negative signs. The

function SQR gives the square root of the number that follows it, in brackets, and there will be an error message if you try to take the square root of a negative number. If you program your square root as SQR(ABS(x)) then, even if x has a negative value, a real quantity will be found and there will be no error messages. While we're on the subject of negative signs, SGN will find the sign of the number that follows it within brackets. The value of SGN will be  $-1$  if the number is negative,  $0$  if the number is zero, and  $+1$  if the number is positive. Figure 4.4 shows this instruction in action. When you type

```

100 CLS
110 LET J=0
120 PRINT TAB 11;"SIGNS AND NUMBERS"
130 PRINT
140 PRINT"Type a number, which can be
positive,"
150 PRINT"negative or zero."
160 INPUT a
170 LET S=SGN(a)
180 IF S=-1 THEN GOTO 260
190 IF S=0 THEN GOTO 290
200 LET R=SQR(a)
210 PRINT "The root is ";R;
220 IF J=1 THEN PRINT "j"
230 LET J=0
240 PRINT
250 END
260 LET a =ABS(a)
270 LET J=1
280 GOTO 200
290 PRINT "The number is zero!"
300 END

```

Fig. 4.4. Using SGN to analyse the sign of a number.

a number, its sign is analysed by line 170, so that the value of S must be  $-1$ ,  $0$  or  $+1$ . Lines 180 and 190 deal with the cases where the number is negative or zero, and line 200 finds the square root of the positive number. If the number was negative, line 180 sends the program to line 260 where the absolute value is found. This removes the negative sign, and the 'flag' variable J is then set to 1. The flag is used to signal that this *was* a negative number, so that the GOTO in line 280 will print the square root of the number and the line 220 will add the letter j. By convention, mathematicians use j (or i) to indicate the imaginary quantity which is the square root of  $-1$ , so that the square root of a number, such as  $-4$  will be written as  $2j$ , meaning the root of  $+4$  multiplied by j. If the number that you typed was zero,

#### 44 *Lynx Computing*

then the program goes to line 290 – we can't take a square root of zero.

### **Logs and antilogs**

Fairly often in technical work, we need to take the *log* of a number. Many computers deal only with the natural (base e) logarithm, but the Lynx will operate with both the natural and the more familiar base 10 logs. For example, if you want to work with amplifier gain figures in units of decibels, then you would use the formula:

$$\text{dB} = 20 \times \log (\text{voltage gain})$$

and this would be programmed as:

$$\text{LET d} = 20 * \text{LOG(g)}$$

using the variable *g* to hold the value of voltage gain. When you need to use the natural logarithm, the instruction word is *LN*. For antilogs, *ANTILOG* gives the base 10 antilog, and *EXP* gives the natural antilog. *EXP(x)* is the quantity that is written in formulae as  $e^x$ , and it occurs widely in equations such as  $A = A_0 e^{-\lambda t}$ , the radioactive decay formula, and in the formulae for the discharging of capacitors and draining water from tanks.

### **Angular antics**

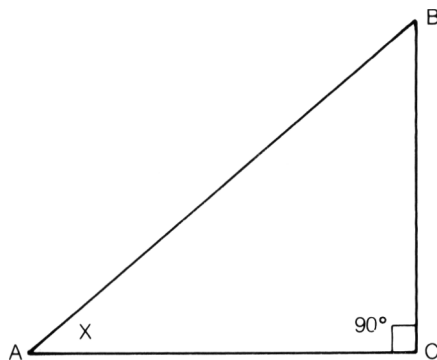
Many of the number functions of the Lynx are a dream for the serious scientific or engineering user, but others also have a wide range of other uses. Among these others are the trigonometrical functions which are listed in Fig. 4.5. A trigonometrical function is the ratio of the lengths of sides of a triangle with one right angle, and with one angle that we can label as *X*. Figure 4.6 illustrates three of these functions for a right-angled triangle. The values of these three functions, *SIN*, *COS* and *TAN*, vary considerably as the angle is changed, and the program of Fig. 4.7 illustrates these changes for a range of 10 to 80 degrees. The Lynx functions *SIN*, *COS* and *TAN* need the angle to be put into units of radians (it makes the calculation of the quantity easier), and *RAD* can be used to perform this conversion. If you need to convert radians to degrees (because functions like *ARCCOS* and *ARCSIN* will give an angle in radians), then *DEG(angle)* will carry out this opposite conversion.

Instruction	Function
RAD	Converts angles in degrees into radians.
DEG	Converts angles in radians into degrees.
COS	Finds the cosine of an angle; angle must be in radians.
SIN	Finds the sine of an angle; angle must be in radians.
TAN	Finds the tangent of an angle; angle must be in radians.
ARCCOS	Finds the angle in radians when the value of its cosine is given.
ARCSIN	Finds the angle in radians when the value of its sine is given.
ARCTAN	Finds the angle in radians when the value of its tangent is given.

*Note:* Angles are measured in degrees or radians.

Both of these quantities are defined in terms of one complete revolution. There are 360 degrees in one revolution, so that one degree is  $1/360$  revolution. There are  $2\pi$  radians in a revolution, so that the radian is about 57.3 degrees.

*Fig. 4.5.* The trigonometrical functions of Lynx.



$$\text{Sine of angle } x = \frac{\text{length of side BC}}{\text{length of side AB}}$$

$$\text{Cosine of angle } x = \frac{\text{length of side AC}}{\text{length of side AB}}$$

$$\text{Tangent of angle } x = \frac{\text{length of side BC}}{\text{length of side AC}}$$

These functions are abbreviated to sin, cos and tan

*Fig. 4.6.* The meaning of SIN, COS and TAN in terms of a triangle.

The ARCCOS, ARCSIN and ARCTAN actions find the angles whose COS, SIN or TAN respectively have the value that follows the function, within brackets. You have to be careful here to avoid

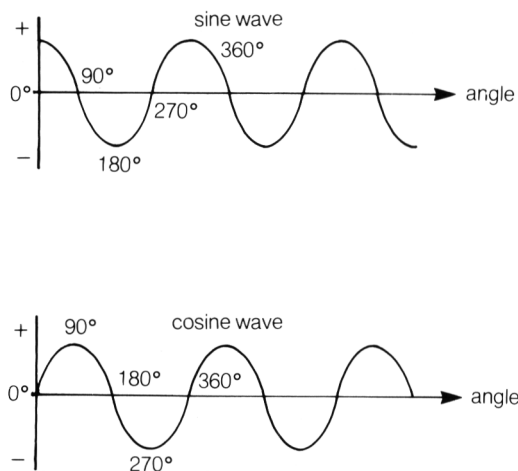
```

100 CLS
110 PRINT TAB 12;"TRIG. FUNCTIONS"
120 PRINT
130 PRINT "Angle";TAB 9;"SIN";TAB 21;"
COS"; TAB 31;"TAN"
140 FOR A=10 TO 80 STEP 10
150   PRINT A; TAB 9;SIN(RAD(A)); TAB
21;COS(RAD(A)); TAB 31;TAN(RAD(A))
160 NEXT A
170 PRINT
180 PRINT"Angle A is in degrees"

```

*Fig. 4.7.* A program which prints a list of values for SIN, COS and TAN of angles.

impossible values. A SIN or COS cannot have a value greater than 1 or less than  $-1$ , and any attempt to find the ARCCOS or ARCSIN of numbers outside these limits will cause the error message – ‘function argument error’. The ‘argument’ is the number that is placed within the brackets.



*Fig. 4.8.* Graphs of sine and cosine of angles. The shapes of the graphs are the same, but the starting positions are different.

Where the graphics use of these functions comes in is that the value of a sine or cosine, plotted for angles that run from  $0^\circ$  to  $360^\circ$ , will give a snake-like curve that is called a sine or cosine wave (Fig. 4.8), depending on which function you used. The shape, in fact, is identical; only the starting point is different. We can make use of this in an example, Fig. 4.9. A range of N from 3 to 360 is used, with a step size that will produce results in a reasonably fast time. The



```

100 CLS
110 PRINT TAB 17;"SINES"
120 PRINT
130 FOR N=3 TO 360 STEP 6
140   PRINT @ N/3,100-50*SIN(RAD(N));"
   *"
150 NEXT N

```

*Fig. 4.9.* A sine wave graph program (later you will be able to use the DOT instruction for this purpose).

values of N are used as values for column numbers in a PRINT@ instruction, and the row numbers are derived from a SIN of the value of N. By using the factor  $100 - 50 * \text{SIN}(\text{RAD}(N))$  we ensure that we print at row 10 (remember that the @ number is ten times the row number) when the sine value is small. By using a negative sign in the expression, we ensure that we print at higher rows when the SIN has a positive value, and at lower rows when the SIN has a negative value. The result is to plot a set of asterisks that follow the shape of a sine wave as it snakes across the screen. Because of the peculiarities of the PRINT@ instruction, the asterisks overlap, and some parts of the wave will wipe out earlier parts, but the shape of the wave is well displayed. Better plotting methods will be dealt with later when we consider the DOT instruction.

```

100 CLS
110 PRINT TAB 10;"CIRCLES AND SPHERES"
120 PRINT
130 PRINT "Please type value of radius."
   "
140 INPUT r
150 PRINT "Area of circle is ";2*PI*r*
   *2;" units."
160 PRINT "Area of sphere is ";4*PI*r*
   *2;" units."
170 PRINT "Volume of sphere is ";(4*PI
   *r**3)/3;" units."

```

*Fig. 4.10.* Using PI in an area and volume program.

One of the excellent features of the Lynx is the number of functions that are particularly useful for scientific or engineering use. One of these is the factorial. The factorial of a number is the number multiplied in turn by each number less than itself, going in steps of -1 down to a final value of 1. For example, factorial 6 is  $6 \times 5 \times 4 \times 3 \times 2 \times 1$ . We can find the factorial of a number in a program by using FACT(a), where a is a number variable name, or

the number itself. We can also use the value of  $\pi$  by typing `PI`, so that the program in Fig. 4.10 will calculate the areas of circles and the surface area and volumes for spheres.

## **DIV, MOD and FRACT**

`DIV` and `MOD` are instructions that operate with numbers to give integer results. An integer is a whole number, containing no fractions, and the use of `DIV` and `MOD` can often allow us to program actions which would otherwise require several instructions. `DIV` carries out division, but gives only the whole number part of the division. For example, `7DIV3` gives 2, because the result of  $7/3$ , which is  $2.\overline{3}$ , has a whole number part which is 2. `MOD` is an instruction which gives a remainder: `7 MOD 3` is 1, because 3 into 7 goes twice and leaves 1 as a remainder.

```
100 CLS
110 PRINT TAB 14;"DIV AND MOD"
120 PRINT
130 PRINT "No. ", "MOD", "DIV"
140 FOR N=1 TO 20
150   PRINT N,N MOD 5,N DIV 5
160 NEXT N
```

*Fig. 4.11.* How `DIV` and `MOD` change over a range of values. Try this with different sets of numbers.

What makes these two instructions useful is the way in which they change as the numbers change. Figure 4.11 shows a list of numbers obtained by taking the numbers 1 to 20 and finding the integer result of dividing by 5 (using `DIV5`) and the integer remainder (using `MOD5`). Note that the remainders cycle in the form 1,2,3,4,1,2,3,4,1,2,... as the number `N` increases, while the `DIV` values remain at one value until the whole number part of the result changes. The change in the `MOD` quantity is particularly useful if you have any quantity that you want to take a restricted range of values during a count (in a `FOR...NEXT` loop, for example). It's particularly useful, for example, when we come to deal with the Lynx colours, which can use number codes in the range 0 to 7. If you use `NMOD8`, then whatever the value of `N`, the number that you get from this will be a valid number code for the Lynx.

`FRACT` is an instruction which performs an action that can be used with `INT`. When we use `FRACT(10.67)`, for example, the result

is .67, the fractional part. This can be useful in accounts (how many pennies?) and also in estimating how much may be lost when INT is used in financial programs.

## Numbers, numbers

The Lynx will, like most computers, accept and print numbers in scientific (or 'standard') form. This means that the number is expressed as a value less than 10 and greater than 1, and multiplied by a power of ten. For example, the number 387.6 can be written as  $3.876 \times 10^2$ , and 0.00012 can be written as  $1.2 \times 10^{-4}$ . The Lynx allows you to type numbers in this form using E in place of the 10, so that we can type 1.7E5 (which is  $1.7 \times 10^5$ , or 170000) or 2.26E-2, which is 0.0226. This can be very useful when you want to enter numbers from data sheets or books when they are printed in this standard form. The Lynx will also print numbers on the screen in this form if the numbers are very large or very small.

One problem that haunts all computers is that of number accuracy. A computer stores a number by converting it into a binary number, one whose base is 2. A binary number consists of a set of 0s and 1s, and it is split into a fractional part and a power-of-two part so that the number can be stored without using too much memory. The conversion is never exact unless the number happens to give a fraction that is a power of 2, like .5 ( $2^{-1}$ ), .25 ( $2^{-2}$ ), .125 ( $2^{-3}$ ) and so on. This means that all computer calculations are liable to be slightly inaccurate. During calculations, your Lynx uses eight digits, but when the number is displayed on the screen its value is rounded, either up or down, to six digits. What you see on the screen is therefore not necessarily identical to what is stored. For a lot of purposes, this is not important, but with many other computers it causes unfortunate effects in programs which compare amounts. You may find, for example, that the computer insists that  $2 - 1$  is not the same as  $3/3$ ! The Lynx, however, is remarkably accurate in its handling of numbers, and it is most unlikely that you will catch it out in this way, unlike some of its competitors.

Also, unlike most other computers, Lynx allows you to see the size of the numbers before the rounding operation. If you type ROUND OFF, the rounding stops and the number that you see on the screen is the same as the number which is stored in the memory. To restore the rounding, type ROUND ON. Try the program of Fig. 4.10, for example, with the lines 105 ROUND OFF and 180

ROUND ON added, and see the difference in the screen presentation.

An instruction that is occasionally useful if you want to work to several places of decimals is TRAIL. If you type TRAIL ON, then each number is printed with the same number of digits. If rounding is on, then all numbers will be displayed with six places of decimals. If rounding is off, all numbers will be displayed with eight places of decimals. This may mean that 2 is displayed as 2.0000000000, but at least you can be certain that it is 2 and not 2.0000000001!

The largest number that the Lynx can use has the value of 9.9999999E+63, and this can be referred to in a program as INF – which is a lot shorter. The quantity INF must not be used while rounding is on, because if this number is rounded up the computer will be unable to deal with it and you will get the ‘number out of range’ error message. INF can be used when you are trying to avoid a number going out of range, by using a comparison like:

```
IF x > INF/2 THEN GOTO 5000
```

but such a comparison must be preceded with a ROUND OFF and followed by a ROUND ON.

## Chapter Five

# Strings and Things

So far, we have made use of string variables and number variables, and Chapter 4 dealt with some of the operations that can be carried out on numbers and on number variables. It's time now to look at string variables, because a large amount of operations can be carried out on string variables that cannot be carried out on number variables. Because of the differences between what we can do with a number variable and what we can do with a string variable, we have instructions which will convert a number stored as a number variable into the form of a string variable, or convert a number stored as a string variable into the form of a number variable.

```
100 CLS
110 PRINT TAB 16; "JOINING"
120 PRINT
130 LET A$="12",B$="13",C$="SMITH",D$=
    "JONES"
140 PRINT A$;" + ";B$;" IS ";A$+B$
150 PRINT
160 PRINT C$;" + ";D$;" IS ";C$+D$
170 PRINT "-we can make this ";C$+"-"+D$
180 PRINT A$;" + ";B$;" is really ";
    VAL(A$)+VAL(B$)
```

*Fig. 5.1.* Concatenation, or joining, of strings.

Just to emphasise the difference, take a look at the program in Fig. 5.1. In line 130 we assign four variables, all as string variables. Lines 140 and 160 then show that the effect of the + sign on string variables is to join them up: "12" + "13" makes "1213", not "25"! The action is the same whether the string consists of numbers or letters, and it can often be useful in putting strings into some desired form, as in line 170. If we want to carry out arithmetic, we cannot do so on numbers that are stored as strings, but the instruction VAL allows us

```

100 CLS
110 PRINT TAB 18;"VAL"
120 PRINT
130 LET A$="26 ROWAN CLOSE"
140 LET B$="ROOM 504"
150 PRINT "The address is ";VAL(A$)
160 PRINT "The room No. is ";VAL(B$)

```

*Fig. 5.2.* Showing that VAL cannot operate on 'embedded' numbers.

to convert. If A\$ = "12", then VAL(A\$) is the number 12 stored as a number, and this quantity can be treated as a number. Line 180 therefore gives the correct result in number form, 25.

VAL will find a number that is followed by letters, but it cannot extract a number from inside a string, as Fig. 5.2 shows. The program can extract the number 26 from the address, but it can't extract the number 504 from the ROOM 504 (am I giving my age away here?).

The opposite of VAL is STR\$. The effect of STR\$ is to convert a number or number variable which is placed in brackets following STR\$, into string form. This allows us to make use of the string operations that appear later in this chapter on a number, and we'll illustrate the use of STR\$ then.

LEN (brother of VAL) is a string action that we can make a lot of use of. LEN(A\$) will find the number of characters in the string, and we can use this, for example, as a way of finding TAB numbers.

```

100 CLS
105 DIM T$(39)
110 PRINT TAB 19;"CENTRE IT"
120 PRINT
130 INPUT "Please type a phrase";T$
140 IF LEN(T$)>38 THEN GOTO 170
150 PRINT TAB 20-LEN(T$)/2;T$
160 END
170 PRINT "That's too long - try a
shorter one."
180 GOTO 130

```

*Fig. 5.3.* A program to centre a phrase in a line.

Figure 5.3 shows a program that will centre any phrase that you type on to a line of the screen. The string length has been dimensioned to 39 to allow for a string that almost fills the line. You type the phrase (your name, perhaps) in line 130, and it is allocated the variable name T\$. In line 140, the length of T\$ is checked, and if it consists of more than 38 characters, you get another chance. If the phrase is reasonably short, then line 150 prints it centred. The TAB instruction uses the expression  $20 - \text{LEN}(T\$)/2$ , which means that the number of characters in T\$ is found, divided by 2, and the result subtracted from 20. This is the same as the formula that we used in Fig. 2.6, but used here as the expression following a TAB. Virtually all instructions that will work with a number variable will work with an expression like this.

We can combine the uses of TAB and LEN to line up numbers in a way that can't be achieved so easily using number variables. Figure 5.4 illustrates the difference. The first part of the program, from lines 110 to 190 uses number variables, and prints a list of them in a

```

100 CLS
110 PRINT TAB 16;"COLUMNS"
120 PRINT
130 LET T=0
140 FOR N=1 TO 5
150   READ J
160   PRINT TAB 10;J
170   LET T = T+J
180 NEXT N
190 PRINT "Sum is ";T
210 PAUSE 50000
220 CLS
230 REM SECOND PART
240 RESTORE
250 PRINT TAB 14;"NEAT COLUMNS"
260 PRINT
270 FOR N=1 TO 5
280   READ J
290   LET J#=STR$(J)
300   LET L = LEN(J#)
310   PRINT TAB 20-L;J#
320 NEXT N
330 LET L=LEN(STR$(T))
340 PRINT"Sum is -"; TAB 20-L;T
350 DATA 1.11.340.4127.50416

```

Fig. 5.4. Lining up numbers in two different ways.

column. The column is lined up at the left-hand side of each number, though, which looks inelegant. Lines 250 to 230 do the same piece of work, but by converting each number into a string we can line up the right-hand sides this time! This is done by using the length of each string and printing at a TAB number equal to  $(20 - \text{length})$  in line 310. By doing this, the TAB portion for a number of many digits is smaller than the TAB number for a number of fewer digits, automatically lining up the right-hand sides. As we've used it here, the method doesn't work for numbers that include a fraction; we have to think further about that one and use some other instructions!

### Slicing strings

You normally think of slicing as an operation that you carry out on string beans. As applied to string variables, it means taking a copy of a selected part of the string starting from some fixed point.

The first of the three string slicing instructions is LEFT\$. You have to follow it by two items within brackets. The first item is the variable name of the string that you want to slice, then there must be a comma. The second item is a number, number variable or expression which specifies the number of letters that you want to slice.

```
100 CLS
110 PRINT TAB 16;"INITIALS"
120 PRINT
130 INPUT "Your first name,please";A$
140 LET N$=LEFT$(A$,1)
150 INPUT"Your surname,please";A$
160 LET N$=N$+"."+LEFT$(A$,1)+"."
170 PRINT "You will be known as ";N$
```

*Fig. 5.5.* Using LEFT\$ to extract initials.

For example, if A\$ = "FELINE", then LEFT\$(A\$,2) is 'FE' – the first two letters, counting the first letter on the left-hand side at number 1. Figure 5.5 illustrates LEFT\$ being used to find a pair of initials. This might perhaps be used to identify a player in a game, or to put a pair of initials on a memo. In line 130, when you type your first name and press the RETURN key, the name is assigned to variable A\$. Line 140 then uses a new variable N\$ to hold the first letter of A\$. This is all we need to use A\$ for, so that line 150 uses the same variable name again to hold your surname. Line 160 then



```

100 CLS
110 PRINT TAB 13;"NUMBER FINDER"
120 PRINT
130 LET A$="ROOM504"
140 LET N=0
150 REPEAT
160   LET N=N+1
170 UNTIL ASC(RIGHT$(A$,N)) > 57
180 LET N=N-1
190 PRINT"Number is ";VAL(RIGHT$(A$,N))

```

Fig. 5.6. Extracting an embedded number by using RIGHT\$.

gathers up the first initial, a full-stop, the second initial, and another full-stop into variable N\$. When N\$ is printed it consists of your two initials with a full-stop after each letter.

The second string slicing instruction is RIGHT\$. When RIGHT\$ is used, the slicing starts from the right hand side of the selected string, and counting also starts from this side. If, for example, B\$ = "STALKING", then RIGHT\$(B\$,4) is 'KING'. RIGHT\$ is not used to such an extent as LEFT\$, but Fig. 5.6 illustrates a way of getting a number extracted from the end of a string – the operation you could not carry out simply by using VAL.

In line 130, A\$ is defined, using the type of embedded number that we could not extract using VAL. Line 140 sets variable N as 0, and lines 150 to 170 check each character of A\$ starting with N=1, at the right-hand side. This introduces a new instruction, ASC. ASC applied to any string, in brackets, finds the ASCII code of its *first* character. Now the numbers 0 to 9 have ASCII codes of 48 to 57 inclusive, and while the ASCII code of the character that we have sliced is 57 or less, it must be a number. We couldn't use VAL to make this selection, because VAL("") is the same as VAL("M"), equal to 0, and the method would not work on a number like this which contained a zero. We would, incidentally, have to tighten up our testing if the string contained characters whose ASCII codes were less than 48. Now, by using a REPEAT...UNTIL loop, we can find the value of N which selects the first letter, reading from the right. The letter 'M' is the fourth along, unless you left a space, so that N will be 4 when it reaches this point, and stops the REPEAT...UNTIL loop. We can now decrease N by 1 in line 180 and then use it to select the number, using VAL this time to get the number from the sliced string which contains no letters to get in the way. Note that this method works even if you leave several spaces between the 'M' and the '5'.

The most useful of all the slicing instructions, however, is `MID$`. `MID$` makes use of two numbers, one to specify the starting letter that you want to use, numbering from 1 at the left-hand side of the string. The other number then specifies how many characters you want to select, starting with this position. For example, if `A$ = "CONCATENATION"` then `MID$(A$,4,3)` gives `CAT`. The `C` of `CAT` is the fourth letter, counting the first `C` as letter No. 1, and we take three letters of the string, starting with this `C` at No. 4 position. Remember, incidentally, that though we talk of ‘taking’ a slice from a string, *nothing is taken out* from any sliced string, we simply *copy* letters from the string.

```

100 CLS
110 PRINT TAB 16;"LINEUP"
120 PRINT
130 LET T=0,J=0
140 FOR N=1 TO 8
150   READ J$
160   REPEAT
170     LET J=J+1
180     UNTIL MID$(J$,J,1)="."
190     LET T$=LEFT$(J$,J)
200     PRINT TAB 20-LEN(T$);J$
210     LET T=T+VAL(J$)
220   LET J=0
230 NEXT N
240 PRINT"Total is ";T
250 DATA .001,.01,.1,1.1,12.24,126,317
    ,1046.7,12.7746

```

*Fig. 5.7. Using `MID$` to print numbers so that the decimal points line up.*

Figure 5.7 shows `MID$` being used to line up a set of numbers so that the decimal points are in a vertical line. Things are starting to get more complicated now, so hang on to your hats. Line 130 sets some variables to zero, and line 140 starts the main loop which will read eight numbers and place them in a column. When a number is read in line 150, the `REPEAT...UNTIL` loop in lines 160 to 180 is used to find how far along the number the decimal point is located. The variable `J` is used for this purpose, and when line 180 finds where the decimal point is, the value of `J` is the number of characters measured from the left-hand side to the decimal point. Line 190 now finds the length of this string up to the decimal point – if the number starts with a decimal point, then this is taken as one character. Line 200 then prints the number with tabulation arranged to keep the decimal point always in the same place along each line. Variable `T` keeps the total in line 210 and the value of `J` is reset in line 220, ready

to deal with the next number. Note that each number is read as a string, because the only time we need a number to be a number variable is when we add it to the total, and we can use VAL for this part. Could you perhaps add to the program so that the total is also printed in the correct place?

We'll make more use of these string-slicing instructions as we go along, but for now we're going to look at some other instructions that affect strings, starting with CHR\$, which we've already used briefly. CHR\$ is an interesting instruction that converts an ASCII code to a character on the screen or in a variable name. The instruction, PRINT CHR\$(65), for example, will produce the letter

```

100 CLS
110 PRINT TAB 15,"QUIZ-TIME"
120 REM
130 PRINT
140 PRINT"WHAT IS THE MOST PLAYFUL CAT
?"
150 PRINT
160 PRINT "PRESS ANY KEY FOR THE ANSWER"
170 LET A$=GET$
180 LET B$=""
190 FOR N=1 TO 10
200   READ J
210   LET B$=B$+CHR$(J+20)
220 NEXT N
230 PRINT B$
240 DATA 51,59,56,50,12,56,69,58,68,13

```

Fig. 5.8. A coded message making use of CHR\$.

A on the screen. This can be useful for coding messages, as Fig. 5.8 shows. The program asks a question. If this were a game, it's certain that you would be tempted to LIST it to find the answer. In this case, however, it's not obvious, because the set of numbers in the DATA line 240 is not obviously a clue. When you give up, the FOR...NEXT loop that starts in line 190 reads each number, and a new string B\$ is built up. What you put on to B\$ is not CHR\$(J), however, but CHR\$(J + 20), which is quite a different letter. This is done for each letter in the loop until B\$ is ready to print. Sorry about that!

On a lot of computers, CHR\$ can be used to convert between upper- and lower-case, one letter at a time. Lynx has an automatic upper-case converter, which will convert all the letters of a string to upper-case. Figure 5.9 shows this particular instruction in use. The

```

100 CLS
110 PRINT TAB 15;"UPPERCASE"
120 PRINT
130 PRINT "Your name,please, using bo
th upper and"
140 PRINT"lower case-";
150 INPUT N$
160 LET M$=UPC$(N$)
170 PRINT "In upper case, it's ";M$
180 PRINT "In lower case, it's ";N$

```

*Fig. 5.9. Converting to upper-case using UPC\$.*

upper-case converter instruction is `UPC$`, which needs to be supplied with the variable name of the string within brackets. `CHR$` has a lot of other uses, however, many of which are peculiar to the Lynx. The ASCII codes run from 32, which is the space that you obtain by using the spacebar, to 127, which on Lynx is half of the copyright sign. Of these characters, some are not completely standard in the Lynx version of ASCII code – particularly codes 94, 96 and 123 to 127.

The numbers from 0 to 31, however, are not standardised, as far as small computers are concerned, in any way and each type of computer has its own way of using these numbers. Lynx uses them to obtain some effects which are very difficult to obtain on other computers, and which are of particular interest to anyone who wants to use foreign language letters or to write equations. You'll see why as we run through the examples.

Each of these effects can be obtained on the screen in two different ways. One way is to type `PRINT CHR$(N)`, where `N` is the number. The other way that is open to Lynx users is the instruction `VDU` which is not part of the BASIC of most computers. `VDU` means Visual Display Unit, and the `VDU` instruction followed by a number code means that the action of that code should affect the screen. The considerable advantage of the `VDU` instruction is that it is much easier to combine a number of effects simply by using a string of numbers separated by commas, following `VDU`. Figure 5.10 shows a list of the codes that can be used with `CHR$` and `VDU` in this way, along with their actions. Note that all of the other ASCII codes can also be used with `CHR$` and `VDU`. What we want to look at now is a set of examples which will show just what these codes do. For the moment, we'll leave the codes that affect the colour of the screen, because we deal with them in more detail later, and we'll start with code 4. The effect of `VDU4` or `PRINT CHR$(4)` is the same as `CLS` – to clear the screen and home the cursor to the top left-hand corner.

Code	Effect
∅	No action.
1	Change INK colour; must be followed by new colour number.
2	Change PAPER colour; must be followed by new colour number.
3	No action.
4	Clear screen; home cursor.
5	Move cursor up 1/10 of a line.
6	Move cursor down 1/10 of a line.
7	Sound a beep.
8	Backspace, erase any character under the cursor.
9	Place cursor in next column, like action of comma.
1∅	Move cursor down one line.
11	No action.
12	No action on production model of Lynx (use 32).
13	Action not straightforward, use 31 instead.
14	Turn cursor on.
15	Turn cursor off (cursor will reappear at end of program).
16	Move cursor to top of screen.
17	No action.
18	Inverse video (exchange INK and PAPER colours).
19	Carriage return/line feed if cursor is <i>not</i> at the start of a line.
2∅	Overwrite off.
21	Overwrite on.
22	Backspace the cursor (does <i>not</i> delete character).
23	Cursor to HOME position (in any window – see later).
24	Double-height characters on.
25	Double-height characters off.
26	Not used.
27	Not used.
28	Move cursor up 3/10 of a line (superscript).
29	Move cursor down 3/10 of a line (subscript).
3∅	Clear to end of current line.
31	Carriage return and line feed (to start of next line).

Fig. 5.10. Codes that are not printed, but can be used with CHR\$ or VDU to achieve useful and interesting effects.

The advantage of using these alternatives to CLS is that sets of instructions can be built up more readily, into strings, for example. Take a look at Fig. 5.11. Line 13∅ builds up a string consisting of codes 4, 1∅ then the word LYNX (note that LYNX has to be in quotes). Code 4 clears the screen, and code 1∅ causes a ‘line feed’,

```

100 CLS
110 PRINT TAB 15;"TEST PIECE"
120 PAUSE 50000
130 LET A$=CHR$(4)+CHR$(10)+"LYNX"
140 PRINT A$;" using CHR$"
150 PAUSE 50000
160 VDU 4,10,76,89,78,88
170 PRINT " using VDU"

```

*Fig. 5.11.* Using some of the codes to achieve special effects.

meaning that it selects the next line down. The message is held on the screen by the PAUSE in line 150, and then the same message is printed using VDU. In this case, the instruction consists of a set of code numbers following VDU, using the 4 and the 10 to position the cursor, and the ASCII codes for L, Y, N and X to produce the message.

```

100 CLS
110 PRINT TAB 13;"CURSOR SHIFTS"
120 PRINT
130 FOR N=1 TO 11
140 READ A$
150 PRINT A$;CHR$(6);
160 NEXT N
170 RESTORE
175 PRINT
176 PRINT
177 PRINT
180 FOR N=1 TO 11
190 READ A$
200 PRINT A$;CHR$(5);
210 NEXT N
215 PRINT
216 PRINT
217 PRINT
220 PRINT"a";CHR$(28);"2";CHR$(29);" +
b";CHR$(28);"3";CHR$(29)
230 PRINT
240 PRINT CHR$(28)"3";CHR$(29);"Li";CH
R$(29);"7";CHR$(28)
300 DATA D,I,A,G,O,N,A,L,Y,N,X

```

*Fig. 5.12.* The line-spacing codes in action.

The codes 5 and 6 have an interesting effect. Each of them moves the cursor vertically – but by only one tenth of a normal character line! We can use these instruction codes, therefore, to cause small changes in the vertical position of anything we print. The usual reason for requiring vertical movement like this is for superscripts

and subscripts – letters or numbers which are written slightly above or below the normal line. For this purpose, three-tenths of a line spacing is more useful, and this amount of movement is achieved by the use of codes 28 (up) and 29 (down). Figure 5.12 illustrates all of these codes in use. The first section, lines 130 to 160, reads letters from the DATA line, but performs a shift upwards after each read. This makes the word seem to be written diagonally. Line 170 restores the data so that it can be used again, and lines 180 to 210 carry out the same actions but using code 5 this time to lower the cursor so that the word slopes in the other direction. Note how this action causes earlier letters to be wiped, the same effect as we saw in the asterisk graph program earlier.

Lines 220 to 240 then make use of the superscript and subscript codes. Line 220 prints the expression  $a^2 + b^3$ , and line 240 prints  ${}^3\text{Li}_7$ , both of which are particularly useful if you are working with scientific or engineering programs.

```

100 CLS
110 PRINT
120 FOR N=1 TO 10
130   VDU 7,42
140   PAUSE 2000
150   VDU 8
160   PAUSE 2000
170 NEXT N

```

Fig. 5.13. An attention-getter routine.

The next code, 7, causes a beep, and code 8 will backspace the cursor to the left and erase any character which happens to be in the way. Take a look at Fig. 5.13, in which these codes are both used to draw your attention to the screen. The loop that starts in line 120 causes a beep to be sounded and an asterisk to be printed each time round. After the asterisk has been printed, the PAUSE in line 140 leaves it on the screen for a moment, and then line 150 deletes it. This should produce enough sound and visual reminders to bring your attention to the machine!

There are several codes that deal with the horizontal movement of the cursor, and we have looked at code 8 already. Code 9 has the same effect as a comma in a PRINT instruction – to move the cursor to the next ‘field’, the next one of the (invisible) columns into which we can imagine the screen divided. Code 32 moves the cursor one space right, which is the action of the spacebar, and code 31 causes a

```

100 CLS
110 DIM A$(20)
120 PRINT TAB 12;"CURSOR MOVEMENTS"
130 LET A$=""
140 FOR N=1 TO 19
150   READ J
160   LET A$=A$+CHR$(J)
170 NEXT N
180 PRINT A$
190 DATA 76,9,89,9,78,9,88,9,45
200 DATA 10,31,76,32,89,32,78,32,88,32

```

*Fig. 5.14.* The cursor-movement codes in action.

line feed (next line) and a carriage return (cursor to left-hand side). Figure 5.14 shows these codes in use, packed into a string A\$ by the use of a loop along with CHR\$. Notice that this string has to be dimensioned like any other string of more than 16 characters. Failure to do this can cause some puzzling effects!

```

100 VDU 4,15,10
110 PRINT "Please type LYNX"
120 VDU 10,32,32
130 FOR N=1 TO 4
140   LET A$=GET$
150   PRINT A$;
160   READ a,b,c,d
170   VDU a,b,c,d
180 NEXT N
190 DATA 31,31,31,9,16,9,32,32
200 DATA 10,10,9,32,10,10,10,9
210 VDU 10,10,10,31
220 PRINT "Bit unsteady, aren't we?"

```

*Fig. 5.15.* Deceiving the eye with an invisible cursor!

The presence of the cursor is sometimes unwanted, particularly, as we shall see later, in graphics programs. Consequently, there are two codes which affect the visibility of the cursor. Code 14 turns the cursor on, and code 15 turns it off. These can be combined with the cursor movement codes to produce some interesting effects, because it isn't obvious where text will appear on the screen when you type. Figure 5.15 shows the sort of thing that can be done, causing the letters that you type to appear at unexpected places. Note that code 31 causes a carriage return and line feed, but without erasing any part of the new line, and code 16 makes the cursor move to the top of the screen and its left-hand side without any erase action.

Code 19 is a rather odd and unusual one. It will cause a carriage



return and a line feed if the cursor is not at the start of the line, but has no effect if the cursor *is* at the start of a line. Codes 20 and 21 deal with overwriting, which means writing one character over another, an action which is impossible with most computers. Code 21 switches overwriting on, and Code 20 switches it off; Fig. 5.16 shows this action in use. This has many applications in adding accents or other marks, and in engineering work, where it's useful to be able to produce the  $\pm$  tolerance sign. Producing these effects on the screen, however, will not produce them on a printer!

```

100 CLS
110 PRINT
120 PRINT
130 VDU21
140 PRINT "Curac";CHR$(22);CHR$(29);", "
    ;CHR$(28)"ao"
150 PRINT
160 PRINT "Tolerance is ";
170 VDU28,43,29,29,22,45,28
180 PRINT ".02 mm."
190 VDU 20

```

Fig. 5.16. Using the overwriting codes.

```

100 CLS
110 PRINT
120 VDU24
130 PRINT TAB 14;"LYNX Titles"
140 PRINT
141 PRINT
142 PRINT
143 PRINT
150 VDU25
160 PRINT".....can be large or small..
    ."
170 FOR N=1 TO 4
180 PRINT
190 NEXT N
200 VDU 24
210 PRINT "END OF MESSAGE"
220 VDU25

```

Fig. 5.17. Using double-height letters for titles.

Code 22 backspaces the cursor, but without deleting a character, and codes 24 and 25 are particularly useful for titles. Code 24 turns on double-height characters, so that everything that is printed following this instruction will be in double-height characters. Figure 5.17 illustrates this in use for titles. Note that the PRINT instruction

```

100 CLS
110 PRINT
120 PRINT "This is a demonstration"
130 PAUSE 50000
140 VDU 28,28,28,5
150 PRINT "LYNX";
160 PAUSE 20000
170 VDU 30

```

*Fig. 5.18.* The text-deleting codes in use.

used by itself will space the cursor much further down the screen when double-height printing is in use.

Codes 26 and 27 are not used at present, and we have previously looked at the actions of codes 28, 29 and 31. That leaves 30, which clears to the end of a line. This is useful if you want to use the same line for several messages, as it prevents a short line from being cluttered by the remaining words from a longer line. Figure 5.18 illustrates an application in which a phrase is printed and after a short delay the cursor is returned to the start of the line and a new word is printed. After another pause, the remaining part of the line is deleted. This can be used in a program that calls for several answers. By wiping the line each time, the screen can remain uncluttered – see the WINDOW instruction later.

## KEY\$ and GET\$

The Lynx contains an unusual number of useful instructions for detecting a key being pressed. We have already made use of GET\$, which causes the computer to wait until a key is pressed, and then assigns the character to a variable name if you have used the form LET A\$ = GET\$. Another string action is KEY\$, which has a similar action, but with no wait. At the instant when the KEY\$ instruction is carried out, if a key is not pressed, the value of KEY\$ is a blank string, "". If a key does happen to be pressed at that instant, however, it will be assigned to KEY\$. We can simulate the action of GET\$ by using KEY\$ in a loop, such as:

```

100 LET K$ = KEY$
110 IF K$ = "" THEN GOTO 100

```

but this action, necessary in many computers, is not needed for Lynx. KEY\$ is particularly useful, however, as a part of a REPEAT...UNTIL loop. Figure 5.19 shows an example of a

```
100 CLS
110 PRINT "Press any key to proceed.."
120 REPEAT
130   VDU 42
140   PAUSE 1000
150   VDU 8,7
160   PAUSE 1000
170 UNTIL NOTKEY$=""
180 PRINT "All done!"
```

*Fig. 5.19.* Using KEY\$ in a loop.

routine which flashes an asterisk and beeps until a key is pressed. With a small change, this routine can be a very useful one for prompting you to make a selection from a number of items on the screen – but that's an action we'll look at later.

## Chapter Six

# Structures, Lists and Menus

Up to now, we've spent a lot of time looking at the coding of programs, meaning that we've been exploring what can be done with the BASIC instruction words of the Lynx. In this chapter, we're going to start paying some attention to *structure* – how programs are designed and organised. This is a much more important and difficult problem, because knowing how to solve a problem with the help of a computer is much more difficult to learn than how to use the programming language. The computer will pick out your errors in the use of BASIC, but it can do nothing to help you get the logic of the program correct.

One of the most helpful aids to program design is to split a program into a set of almost independent parts. In this way, you need only design one bit at a time, and it's also possible to keep 'standard' pieces of program which can be attached to any other program by making use of the RENUM and APPEND commands of the Lynx. Before we get involved in program design, though, we'll take a look at the idea of breaking a program into small sections.

Suppose we have a program in which the user frequently has to make a Y/N choice by pressing the Y or N keys. You could, of course, type the program lines that carry out this action at each place in the program. Such repeated typing is a waste of time and computer memory, though, and all computers provide for ways of making such repeated typing unnecessary. One of these ways is the use of a *subroutine*.

A subroutine is a piece of program which is designed to carry out an action, but whose last line to be executed contains the instruction word RETURN. That instruction RETURN is a very important one, because it ensures that when this piece of program is completed, the computer then returns to the point where the subroutine was requested. The request, or 'call' is done by using the instruction GOSUB.

```

100 GOSUB1000
110 LET T$="THIS IS A TITLE"
120 GOSUB 2000
130 LET T$="AND SO IS THIS"
140 GOSUB 2000
150 END
1000 CLS
1010 PRINT
1020 RETURN
2000 PRINT TAB 20-LEN(T$);T$
2010 RETURN

```

*Fig. 6.1. Using a subroutine.*

Figure 6.1 illustrates this, using as subroutines the sections that start in lines 1000 and 2000. The subroutine at line 1000 simply clears the screen and prints one line down. The subroutine at line 2000 will print the string T\$ so that it is centred on a line. By using GOSUB1000 in line 100, we cause the program to go to line 1000, carry out this line and line 1010 and then return to line 110. This automatic return that we get when we use GOSUB is particularly useful – if we used GOTO we would need one GOTO1000 in line 100 and a GOTO110 in line 1020. Even more useful is the fact that the return will always be to the line following the GOSUB line, wherever that may be. In the example of Fig. 6.1, GOSUB 2000 is used twice. The first call is from line 120, and it then returns to line 130. The second call is from line 140, and it returns to line 150. This action could not be obtained by using GOTO, except with rather complicated programming, and it makes the action of the program much easier to trace. You know, as you read through a program, that whenever a GOSUB5000 (for example) causes a subroutine to be carried out, the program will return and continue on the next line when the subroutine has finished its action.

An important feature that Fig. 6.1 brings out is the use of END in line 150. Try removing this line (type DEL150, press RETURN) and see the effect when the program runs now. Instead of ending after line 140 the program then goes on to use line 1000, since the normal action of the computer is to perform lines in order. When the word RETURN in line 1020 is met, however, the computer cannot proceed, as it has no record of a GOSUB having been used on this occasion. The computer then stops with the error message 'RETURN without GOSUB'. This is a 'crash-through', and it can be avoided by placing subroutines after a final END instruction so that the subroutines can be used only by means of a GOSUB.

The rules for subroutines are therefore that they must be placed so

```

100 CLS
110 PRINT "TYPE Y OR N (Hold key down!)"
120 GOSUB 1000
130 END
1000 PRINT "*";
1010 LET K$=KEY$
1020 IF K$="Y" OR K$="N" THEN RETURN
1030 PAUSE 3000
1040 PRINT CHR$(8);
1050 PAUSE 3000
1060 GOTO 1000

```

*Fig. 6.2. A Y or N subroutine.*

that they cannot be used except by a GOSUB, and that they have to end with a RETURN. This does *not* mean that RETURN has to be the last line of a subroutine, as Fig. 6.2 illustrates. This is a subroutine for obtaining a Y or N reply – the listing shows also a small piece of program that will call the subroutine, for the sake of demonstration. While the computer waits for a reply, an asterisk flashes, and the program will return as a result of the test in line 1020 only if you press the Y or N (not y or n!) keys. How could you ensure that the y or n keys would have the same effect as Y or N?

The Lynx permits you some luxury extras on subroutines that are not available on the run-of-the-mill computers. One is the use of an expression as the number following GOSUB. You can, for example, use GOSUB 1000\*V, as Fig. 6.3 illustrates. This program is one that

```

100 GOSUB 500
110 LET T$="MENU"
120 GOSUB 600
130 PRINT
140 PRINT"1. HEADS."
150 PRINT"2. TAILS."
160 PRINT" Please choose by number."
170 LET A$=GET$
180 LET V=VAL(A$)
190 GOSUB 1000*V
200 END
500 CLS
510 PRINT
520 RETURN
600 PRINT TAB 20-LEN(T$)/2;T$
610 RETURN
1000 PRINT "Yes, heads !"
1010 RETURN
2000 PRINT "Yes, tails !"
2010 RETURN

```

*Fig. 6.3. The GOSUB line number can be calculated from an expression.*

was devised to show the principle, it's not exactly a useful program in its present state. Two choices are printed by lines 140 and 150, and you are asked to choose by typing a number. We could use GETN for this, but using GET\$ will return the character, while GETN will return the ASCII code. Line 180 then extracts the number value from the string, and in line 190 this is used to control the GOSUB. We can also use expressions like  $1000 + V$  or  $1000 - V$  or  $V * 1000 - 200$  in a GOSUB. Note that the example fails if you press any key but 1 or 2. Could you write a few lines between 180 and 190 that would prevent any values other than 1 or 2 reaching line 190? Such a piece of program is called a *mugtrap*, and mugtraps are needed wherever a choice of this type could possibly go wrong. A good mugtrap should trap a mistake, print a message on the screen to explain why it is a mistake, and give the user another chance to enter something more acceptable.

```

100 GOSUB LABEL clear
110 LET T$="TITLE"
120 GOSUB LABEL centre
130 PRINT
140 PRINT "The subroutines are labelled
"
150 END
1000 LABEL clear
1010 CLS
1020 PRINT
1030 RETURN
2000 LABEL centre
2010 PRINT TAB 20-LEN(T$)/2;T$
2020 RETURN

```

Fig. 6.4. A GOSUB using a label name – a special feature of Lynx.

There is another version of GOSUB that is unusual in the Lynx, the ability to use a 'label'. A label is a letter or word which the computer can recognise as being the start of a subroutine. Figure 6.4 illustrates this, using subroutines that are already familiar to you. This time, though, the GOSUB is to a label, with a name following the label. Where the subroutine starts, the word LABEL is used, along with the same label name. This method has two important advantages. One is that line numbers do not have to be used following GOSUB, so that if you change line numbers the subroutines are not affected. If you change line numbers by using the RENUM command for the whole program, this does not affect either type of subroutine, because the line numbers following GOSUB are changed as well. If you change only a few line numbers by editing, though, you may find that you have changed your

GOSUB line numbers. In addition, if you keep a tape of useful subroutines which you can add to your programs with the APPEND command, it's more useful to use labels because it can save a lot of renumbering. Another advantage of using a label name is that the name can be chosen to remind you of what the subroutine does. On other computers this has to be done by having a REM line. REM can also be used on the Lynx, as we saw in Chapter 1. A line that starts with REM will be ignored by the computer when a program runs, but the REM line is printed out in a listing as a reminder to you.

A subroutine has its uses, particularly for carrying out actions like clearing the screen, getting the value of a key, the type of action that we have illustrated. A subroutine, however, is distinctly less useful when we have to pass a variable to it. Take, for example, the familiar subroutine that prints a title centred on a line. In our example, we have used T\$ for the title, and this variable name will have to be assigned to whatever words we want to centre. If we have a set of words that are held as a variable Z\$, then to make use of our subroutine, we have to 'pass the variable', using a line such as:

LET T\$ = Z\$

before we can use the GOSUB. This restricts the usefulness of subroutines, because a subroutine which deals with several variables might require a lot of re-assigning before it could be used.

```

100 PROC clear
105 LET T$="PROCEDURES"
110 PROC title(LEN(T$))
120 PROC space(3)
130 PRINT "We can call these by name"
135 LET T$="END"
140 PROC title(LEN(T$))
150 END
500 DEFPROC clear
510 CLS
520 PRINT
530 ENDPROC
600 DEFPROC title(X)
610 PRINT TAB 20-X/2;T$
620 ENDPROC
700 DEFPROC space(N)
710 FOR J=1 TO N
720 PRINT
730 NEXT J
740 ENDPROC

```

*Fig. 6.5. Using procedures to pass variables.*



The Lynx can overcome the variable passing problem to a limited extent. The keyword here is PROC, which is short for PROCedure. The equivalent of writing a subroutine is defining a procedure, and the equivalent of a GOSUB instruction is the PROC instruction. Procedures are always referred to by name, and we can follow the name with a list of the variables that we want to be passed to the procedure. Procedures are most useful when either number or string variables can be passed, but the Lynx permits only number variables to be passed in this way – alas! Figure 6.5 shows a simple illustration, picked for simplicity rather than usefulness, which uses three procedures. Notice, first, how the use of the procedures makes the main part of the program in lines 100 to 150 very simple and straightforward. By using procedure names that mean something to the user, we can make the program easier to follow as we read it.

In line 100, the use of PROCclear causes the computer to start hunting for the definition of PROCclear, which it finds in line 500. Each definition must start with DEFPROC, and then contain the name. The actions of PROCclear follow in lines 510 and 520, and the word ENDPROC performs much the same action for the procedure as RETURN does for a subroutine. For this particular type of action, then, the procedure doesn't really offer any advantage as compared to a subroutine.

The procedure that is used in line 110 is rather different, though. Its name is PROctitle, and this is followed, within brackets, by a number, the TAB number. This is the number that will be used by the procedure to centre the title, and this illustrates the way in which the number is passed to the procedure. As before, the computer looks for the definition of the procedure, which it finds in line 600. Now in this line, the number that follows DEFPROctitle is not 20 – LEN(T\$)/2, but X. X is called the *formal parameter*; it's the variable that we use in the procedure. Whatever variable or number we have in brackets following PROctitle in line 110 will be reassigned automatically to this X, so that the line 610 which prints T\$ centred will operate with whatever number we send to it. It's an important point to make from an unimportant routine (here again, a subroutine would be easier), because it starts to show how the use of procedures can be more valuable than the use of subroutines.

PROCspace in line 120 really clinches the point. In this example, the number, 3, is passed to a procedure that is defined in line 700. The procedure uses a variable N in a FOR...NEXT loop to print spaces. In the example, the procedure will print three blank lines, but it will print as many lines as we care to pass to the procedure. The

```

100 PROC clear(2)
105 LET T$="SIDE OF TRIANGLE"
110 PROC title(LEN(T$))
120 PRINT "Please type sizes as indicated-"
130 INPUT "First side- ";x
140 INPUT "Second side- ";y
150 INPUT "Angle between sides in degrees-";z
160 PROC tri(x,y,z)
170 PRINT " Third side is ";c
180 END
200 DEFPROC tri(a,b,A)
210 LET c=SQR(a**2+b**2-2*a*b*COS(RAD(A)))
220 ENDPROC
230 DEFPROC clear(N)
240 CLS
250 FOR J=1 TO N
260 PRINT
270 NEXT J
280 ENDPROC
290 DEFPROC title(X)
300 PRINT TAB 20-X/2;T$
310 ENDPROC

```

*Fig. 6.6.* A procedure which passes three number variables.

value assigned to N can also be used in a later part of the program, after the procedure has run.

We can pass more than one number variable to a Lynx procedure. We can, for example, use PROCtri(a,b,A) in which three number variables are passed to a procedure. The variable names must be separated by commas, and the variables that are passed to the procedure must be in exactly the same order as the variables that are used in the definition. Figure 6.6 shows an example of a procedure which finds the size of the third side of a triangle, given the two other sides and the angle between them. In the example, the quantity which is calculated, the third side, has to be 'passed back' from the procedure. Lynx procedures are simple in this respect – all variables that are passed to a procedure or given a value in the procedure are passed back, and their values can be used in the main program again.

## Top-down design

Procedures and subroutines allow us to make effective use of a

method of designing programs that is called 'top-down' design. The principle is a very simple one – we start by designing a skeleton program, containing only the bare bones of our ideas. All the detail which puts flesh on to this skeleton is left to subroutines or procedures rather than tackled right away.

As usual, an example is by far the easiest way of demonstrating how this can be done. Suppose we want to devise a program that allows us to type in a set of examination marks and then print the lowest mark, the highest mark, and the average value. How do we go about it? What we must not, on any account, do is sit down at the keyboard and start typing lines of BASIC. Trying to work out a program in your head is possible if you have enough experience – but for most of us it results in a program that is as useful and as clear to follow as a crate of eels. We need to acquire the habit of planning a program so that it has a sound structure. If we built houses the way some people design BASIC programs, we might start with the plumbing, move to the roof, and think about foundations about half-way through!

<i>Screen</i>	<i>Inputs</i>
Title and instructions	–
Prompts	Marks (repeat)
Results	–

*Fig. 6.7. A diagram to aid program design for the example.*

The first step should be to decide what we expect to see on the screen, and what we should have to type on the keyboard. Figure 6.7 is a simple diagram that reminds you of these items. You would expect to see a title and some instructions on the screen at the start. Following that, you will expect to find a prompt (like Mark No. 1?) which will remind you that an input is required. This prompt will have to be repeated for each entry, so that some sort of loop is called for. Finally, the screen will clear and show you the results.

That's the main bones of the program, and we now have to start filling it out. Some decisions are needed at this stage. Do we, for example, want to put the instructions in as part of the main section of the program, or do we want to place them in a subroutine or procedure? If the instructions are placed in a subroutine or procedure, they can be called up as many times as we like in the

course of the program. In a simple program like this, once-and-for-all instructions are sufficient, but as we shall see, there are still advantages to putting them into a subroutine.

---

```
Start loop, start count.  
Show number.  
Enter item.  
Check for error.  
Check for maximum.  
Check for minimum.  
Add to total.  
Repeat until end indicated.
```

---

*Fig. 6.8.* Listing what has to be done in the main loop.

Next we have to think of the most important section of the program, the piece that deals with the entry and checking of the marks. This is a section that can be designed by itself, and we could approach it as Fig. 6.8 indicates. This section needs a loop, with a counter variable. For each item we can then show the number of the item. This is a useful error-check – if you know that you had to enter 20 items and you find yourself making an entry for item 21, then something is wrong. As each item is entered, we have to carry out four operations. One is to check for silly entries. For a set of exam marks expressed as percentages, a silly answer would be anything below 0% or anything above 100%. You will have to decide for yourself what you will reject and what you will accept in your own programs. Some errors, particularly with numbers, are easy to trap; others, particularly with names, are not.

The second check is for the maximum value. You have to set a variable which will be used to carry this maximum value, and test it against each item that is entered. If any item that is entered has a value greater than the value of this variable, then the variable is reassigned to have the new higher value. Provided that the value of this variable starts at 0, and that each entered value is compared with it, it will end up having the maximum value of the set of numbers. The check for minimum value is done in a very similar way, but this time the variable starts with a value of 100, and each time an entry turns out to have a smaller value, the variable is reassigned to this smaller value.

As each item is entered, its value is added to a total, and the variable which is used for the total must start with a value of zero.

Values should be added to the total only if they have been passed as acceptable – there wouldn't be much point in checking for items like -5% or +115% if we then went ahead and added them in! The whole entry procedure is repeated until we indicate that it is to end. We then have to decide what will indicate an end. We could use a line like:

```
UNTIL GET$ = "s"
```

but this is clumsy because it requires us to press another key just to make the program accept another value each time. A better method would be to kill two deer with one lynx, and make the entry routine end when a negative number is entered. If we do this, then we have to be sure that this value is not rejected by the error routine, and is not taken as the minimum or added to the total.

```
100 GOSUB LABEL cls
110 GOSUB LABEL inst
120 PROC entry
130 PROC print
140 END
```

Fig. 6.9. The main program (core program) – just five lines!

Now we can start to design the main section of the program. It might look as in Fig. 6.9. Do I hear cries of 'cheat'? The main section of the program consists of just five lines which show the order in which the main subroutines and procedures are carried out. This is the *essence* of top-down programming. You leave the detail to the end. This is a perfectly adequate main 'core' for the program. It's easy to follow and easy to check, and it makes use of at least one subroutine that we know very well already – the clear-screen routine.

What we have to do now is to design the subroutines and procedures. These don't need any new methods – you design each one of them as if it were a small program in its own right. Of the routines in Fig. 6.9, the one that we most want to concentrate on is PROCentry, so we'll take a look now at how this procedure might be designed. We have already, in fact, done most of the donkey-work in Fig. 6.8. I have shown a suggested routine in Fig. 6.10. We know that the entry must be repeated, so the procedure starts with the REPEAT instruction. The items are to be counted, so a counter variable must be used, and N looks as good as any name for this purpose. At this point, though, we have to make a note that N has to have a starting value of 0. At some point before all the action starts, then, there must be a line which contains LET N=0. We have left

```

1000 DEFPROC entry
1010 REPEAT
1020   LET N=N+1
1030   PRINT "Item ";N;" is ";
1040   PROC input(J)
1050   PROC max(J)
1060   PROC min(J)
1070   PROC tot(J)
1080 UNTIL SGN(J)=-1
1090 ENDPROC

```

*Fig. 6.10.* The entry procedure. Some actions are 'farmed out' to other procedures.

lines up to 100 free just for such purposes, so we can put this instruction into line 50 before we forget it.

The next action is the prompt, printing the number of the item and asking for an input. This can be combined with a test, because if the input is incorrect, it's easy to go back for another input when the input and test portions are in the same routine. We'll write that one later! We also need a PROCmax to find the maximum value and a PROCmin to find the minimum value. This latter one needs some care because, if we use a negative number to end the entry, the negative number will be assigned as the minimum value unless we exclude negative numbers from the minimum test. PROCtot, which finds the total must also reject a negative value. The end of entry occurs when a negative value is entered because of the use of SGN(J), which will have a value of -1 when any negative number is entered. We can now design the new procedures for this part of the program.

```

2000 DEFPROC input(J)
2010 INPUT J
2020 WHILE J>100
2030   PRINT "Incorrect - exceeds 100"
2040   INPUT J
2050 WEND
2060 ENDPROC

```

*Fig. 6.11.* The procedure for rejecting an oversize entry.

A procedure contained within another procedure is said to be 'nested', just as FOR...NEXT loops are nested, and the first of these nested procedures is PROCinput. We need it to input a number, and check if the number exceeds 100%. If it does, we must point out the error, and ask for another entry. Figure 6.11 looks as if it will cope with this requirement. The input is made, and the WHILE...WEND

loop carries out the checking. If J is less than 1000 or is equal to 1000, then the entry is accepted, but each time a value of more than 1000 is entered, an error message (our own one, not a computer one!) is printed and another input is requested. We check only for excessively large values, because we want to use negative values for ending the program.

```
3000 DEFPROC max(J)
3010 IF J>x THEN LET x=J
3020 ENDPROC
```

*Fig. 6.12.* The maximum value finding procedure.

Next on the list is finding the maximum. We use a number variable x here, which must start with a value of 0. Reminder – type a line 60 which contains LET x = 0. The maximum-finding procedure then reads as in Fig. 6.12. For finding the maximum, we need a variable which starts at a high value – the maximum possible value of 1000 sounds reasonable. Now we have to type a line 70 which contains the instruction LET m = 1000. The minimum-finding procedure (Fig. 6.13) is slightly more complicated because we have to reject a negative value, and this is the purpose of the first test. If SGN(J) is -1, then the procedure ends, and the previous value of minimum quantity is retained.

```
4000 DEFPROC min(J)
4010 IF SGN(J)=-1 THEN GOTO 4030
4020 IF J<m THEN LET m=J
4030 ENDPROC
```

*Fig. 6.13.* The minimum value finding procedure.

```
5000 DEFPROC tot(J)
5010 IF SGN(J)=-1 THEN GOTO 5030
5020 LET T=T+J
5030 ENDPROC
```

*Fig. 6.14.* The totalling procedure.

The totalling procedure in Fig. 6.14 is similar. Once again, we need to have a line early in the program which sets the total value T to zero, and we need to ensure that a negative quantity is not added to the total in the procedure. This exhausts the requirements for procedures within PROCentry. We can now write the final printing procedure, which is illustrated in Fig. 6.15.

Finally, all of the program lines can be entered and tested – the final version is shown in Fig. 6.16. For testing, we need a dummy set

```

6000 DEFPROC print
6010 CLS
6020 PRINT
6030 PRINT "The total is ";T
6040 PRINT "The average is ";T/N;" for
";N;" items."
6050 PRINT "The minimum value was ";m
6060 PRINT "The maximum value was ";x
6070 PRINT
6080 ENDPROC

```

*Fig. 6.15.* The printing procedure.

```

50 LET N=0
60 LET x=0
70 LET m=100
80 LET T=0
100 GOSUB LABEL cls
110 GOSUB LABEL inst
120 PROC entry
130 PROC print
140 END
500 LABEL cls
510 CLS
520 PRINT
530 RETURN
600 LABEL inst
610 PRINT "Enter numbers between 10 and
100."
620 PRINT "Terminate by typing a negative
No."
630 RETURN
1000 DEFPROC entry
1010 REPEAT
1020   LET N=N+1
1030   PRINT "Item ";N;" is ";
1040   PROC input
1050   PROC max(J)
1060   PROC min(J)
1070   PROC tot(J)
1080 UNTIL SGN(J)=-1
1090 ENDPROC
2000 DEFPROC input
2010 INPUT J
2020 WHILE J>100
2030   PRINT "Incorrect - exceeds 100"
2040   INPUT J
2050 WEND
2060 ENDPROC
3000 DEFPROC max(J)
3010 IF J>x THEN LET x=J
3020 ENDPROC
4000 DEFPROC min(J)

```



```

4010 IF SGN(J)=-1 THEN GOTO 4030
4020 IF J<m THEN LET m=J
4030 ENDPROC
5000 DEFPROC tot(J)
5010 IF SGN(J)=-1 THEN GOTO 5030
5020 LET T=T+J
5030 ENDPROC
6000 DEFPROC print
6010 CLS
6020 PRINT
6030 PRINT "The total is ";T
6040 PRINT "The average is ";T/N;" for
";N;" items."
6050 PRINT "The minimum value was ";m
6060 PRINT "The maximum value was ";x
6070 PRINT
6080 ENDPROC

```

*Fig. 6.16.* The complete program for finding average mark, maximum mark and minimum mark.

of entries, preferably a fairly small number, and including at least one value that exceeds 100. We need to keep the number small so that we can see the maximum and minimum value for ourselves and calculate our own average. This is essential because, otherwise, we shall not notice if an incorrect value is added in because of the failure of one of our tests. We can then enter our data and see what the computer makes of it.

Finding faults? That's a subject by itself, and we'll have to defer it to Chapter 10.

## Chapter Seven

# Number and String Lists

### Arrays arise!

Up to now we have not dealt with many variable names in any one program. The average-mark program whose design occupied us in Chapter 6, for example, did not store the value of each mark, it simply added each one in to a total as it was entered. The problem we have to look at now is what to do if we want to keep individual values of large numbers of items. We have only 52 variable names available for numbers, and it would be very clumsy to attempt to assign a different variable name to each of a large number of items.

This problem is dealt with by a different type of variable – the *array*. An array is a list of items, and what distinguishes one item from another is not the name but a ‘label number’, called a *subscript*. In printed work, we often refer to a set of related values as  $A_1, A_2, A_3$  and so on, and the computer permits us to work with something similar. If we use a variable name  $A$ , then different items can be assigned to  $A(1), A(2), A(3)$ , and so on. These names are pronounced ‘A-of-one’, ‘A-of-two’, ‘A-of-three’, etc. The number is called the subscript number, and each separate ‘name’ like  $A(2)$  is called an element of the array.

The Lynx allows the same choice of letters for number arrays as it does for simple number variables. An array item, furthermore, is not confused with a simple variable item, so that we can have  $a, A, a(1), A(1)$ , all representing different values. Now we have to see what we can do with arrays and how we can fill an array with values.

The usefulness of an array arises because the subscript need not be a number. It can be a number variable or an expression which works out to a number value. If we have an array of values  $A(1)$  to  $A(100)$ , for example, we can pick out any one of these values by using a routine such as:

```

PRINT"Which one would you like";
INPUT N
PRINT "Item ";N; " is ";A(N)
PRINT"The next item is ";A(N+1)

```

with the last line demonstrating that we are not confined to using just the variable name. The use of arrays opens up new prospects for programming, but before we can make much use of arrays, we have to know how to prepare the computer for the use of an array.

The 'preparing' instruction is DIM. We have met DIM already in connection with the number of characters in a string variable, and the same instruction word is used to notify the computer that an array is being used, and what maximum value of subscript number you are likely to use in an array. If we type:

```
DIM x(20)
```

for example, we are instructing the computer to lay aside enough memory to store numbers of an array which can use subscript numbers up to 20. That's a total of 21 subscripts, because we can use  $x(0)$  as well as  $x(1)$ ,  $x(2)$ , ...,  $x(20)$ . Using DIM  $x(20)$  doesn't commit us to having to use 20 items, it simply means that we must not exceed that number of subscript. If you do exceed this number at any stage, you will get the error message, 'subscript out of range' to remind you. Once you have 'dimensioned an array' by using an instruction such as DIM  $x(20)$ , then you must *not* attempt to change the dimensioning of the same array in the same program. This can't be done without a complete spring-clean of the Lynx memory, which means losing everything that you have stored in the memory. Any attempt to change the dimensioning of an array during a program therefore calls up a 'redimensioned array' message and the program stops. You can, of course, change your dimensioning instruction before the program runs; what you can't do is change it by a line in the program as the program runs. We can, incidentally, dimension more than one array with one DIM line. We can type DIM  $a(70)$ ,  $b(30)$ ,  $c(50)$ ,  $d(15)$  all in one line, using commas to separate the different arrays.

Now let's take a look at the filling and use of arrays. Because the array subscript can be a variable name, a FOR...NEXT loop is the most obvious way of filling an array. Figure 7.1 shows such a loop which controls READ instructions to put ten numbers into an array  $x$ . The lines 140 to 160 read numbers into an array from the DATA line 270. The instructions in lines 170 to 200 ask you to type two

```

10 DIM x(10)
100 CLS
110 PRINT
120 PRINT TAB 14;"NUMBER ARRAY"
130 PRINT
140 FOR N=1 TO 10
150   READ x(N)
160 NEXT N
170 PRINT "Please type two numbers."
180 PRINT "The program will pick out ar
ray items"
190 PRINT "that lie between these lim
its.Type the"
200 PRINT "larger number first."
210 INPUT A,B
220 IF B>=A THEN GOTO 170
230 FOR N=1 TO 10
240   IF x(N)<=A AND x(N)>=B THEN PRINT
x(N)
250 NEXT N
260 END
270 DATA 1,5,25,80,110,200,500,1000,4000,
6000

```

*Fig. 7.1. Filling an array and then choosing values.*

numbers, larger one first, and line 220 following the input will return the program to line 170 if the numbers are equal or in the wrong order. In this example, you need to choose numbers that are considerably separated! The choice is made in lines 230 to 250, by comparing each item of the array with the two numbers that you have entered, and printing the numbers that pass the test in line 240. If we had used a subroutine here:

IF x(N)<= A AND x(N) >= B THEN GOSUB LABELprint  
then we could have arranged a 'flag' variable to be set so that a message like NO NUMBERS BETWEEN 30 AND 28 FOUND is printed.

An array also makes it possible to sort numbers into ascending or descending order. The methods that are used for this purpose are not so simple as the methods that we use for selecting an item. Also, the methods that are simplest to explain are unfortunately the least useful, because they take a long time when a large number of items have to be sorted. One of the more efficient routines for sorting numbers is called the Shell-Metzner sort, and Fig. 7.2 illustrates a version of this for the Lynx. The numbers are entered in a main routine, using INPUT so that you can use whatever numbers you like – as written, the program is dimensioned for 20 numbers. If you

```

10 DIM a(20)
100 CLS
110 PRINT
120 PRINT TAB 14;"Number Sort"
130 PRINT
140 PRINT "Please enter your numbers-"
150 FOR N=1 TO 20
160   PRINT "Item ";N;" ";
170   INPUT a(N)
180 NEXT N
185 LET N=N-1
190 PRINT "Now to sort them...."
200 GOSUB LABEL sort
210 CLS
220 FOR N=1 TO 20
230   PRINT "Item ";N;" ";a(N)
240 NEXT N
250 END
300 LABEL sort
310 PRINT "Sorting...please wait"
320 LET Y=1
325 REPEAT
330   LET Y=2*Y
340 UNTIL Y>=N
350 LET Y=INT((Y-1)/2)
360 IF Y=0 THEN GOTO 440
370 LET T=N-Y
380 FOR I=1 TO T
390   LET J=I
400   LET Z=J+Y
410   IF a(Z)<a(J) THEN GOTO 450
420 NEXT I
430 GOTO 350
440 RETURN
450 SWAP a(Z),a(J)
480 LET J=J-Y
490 IF J>0 THEN GOTO 400
500 GOTO 420

```

Fig. 7.2. A Shell-Metzner sorting routine for the Lynx.

want to use a different range you will have to change the DIM line and the size of the number in the FOR...NEXT loop. The actual sort is operated by splitting the list into roughly equal parts, and going through the parts, swapping numbers that are in the 'wrong' part. 'Wrong' in this sense means that all the lower numbers should be in the first part and all the higher numbers in the second part. Once this is completed and the numbers have been reallocated as needed, the parts are themselves split into sections and the arrangement is done again. When each number is in its correct place, the subroutine returns. This example puts the numbers into ascending order. Can

```

100 DIM B(50)
110 FOR Q=1 TO 50
120   LET B(Q)=Q
130 NEXT Q
135 LET Q=Q-1
140 LET F=0
150 PRINT
160 PRINT "What number would you like";
170 INPUT A
180 GOSUB LABEL find
190 IF F=1 THEN PRINT "This is item ";N
200 ELSE PRINT "Lies between items ";
   N-1;" and ";N
210 END
1500 LABEL find
1505 LET c=1
1510 LET J=Q+1
1520 LET a=J
1530 LET b=INT((a+1)/2)
1540 IF b=0 OR b>Q THEN GOTO 1640
1550 IF A<B(b) THEN GOTO 1600
1560 IF A=B(b) THEN GOTO 1620
1570 LET a=b+J
1580 LET c=b
1590 GOTO 1530
1600 IF a=b+c THEN GOTO 1630
1605 LET a=b+c
1610 GOTO 1530
1620 LET F=1
1630 LET N=b
1640 RETURN

```

*Fig. 7.3.* A routine which will find the position of an item in a sorted list. The low-numbered lines create a sorted list for testing the program.

you see how it could be altered to put the numbers into descending order?

One of the great advantages of having an 'ordered' list of this type is that it's much easier to find a number in such a list. If you want to find which item in the list has the value 36, or which item is closest to 36, there is a very efficient 'finding' subroutine which operates as illustrated in Fig. 7.3. This routine will have to be added on to your number list and sort routine, because it has to work on a sorted list. As shown, the subroutine is attached to a few lines which generate a set of consecutive numbers. The subroutine works by splitting the list into two equal (or almost equal) parts. If the number that you have specified is greater than the midway number at which the lists split, then your number must be in the second part, otherwise it is in the first part. You might, of course, be lucky and find that your

chosen numbers is in the middle! If your number is in one of the split parts, then the part in which it is located is split again, and the position narrowed down to one of the two pieces again. By repeating this procedure, your number will be found after only a few comparisons. This is very much faster than looking through the whole list, comparing your choice with each item, as you would have to do if the list were not in order. In the routine of Fig. 7.3, if the item that you have specified is not in the list, then the program prints the numbers that lie just above it and below it.

## String arrays

For anyone whose interest in computing is not biased to the scientific, engineering or mathematical side, string arrays are probably of considerably more interest than number arrays. As the name suggests, a string array is a list of strings, each one using the same variable name but with a different subscript. The manual that went out with the early Lynx models made no mention of string arrays or how to dimension them, so that this information may be useful to many readers.

To start with, a string array must use the same type of name as any ordinary string variable. That means one of the upper-case letters. This has then to be followed by a subscript number enclosed in brackets, as you might expect after having seen number arrays. The big difference comes in the method of dimensioning. Normally, a string variable has to be dimensioned for its maximum length, so when a string array is dimensioned, an extra number is needed. This extra number is, of course, the maximum subscript number that you will want to use. Suppose, for example, that we want to dimension an array to take strings of up to 25 characters, and to use subscript numbers of up to 50. The dimensioning line will read:

```
DIM S$(25)(50)
```

placing brackets round *each* number and with no commas. This is rather an unusual form of instruction, and one which does not resemble the methods that are used on other computers, so perhaps it's a good idea to illustrate it in use. Figure 7.4 is an illustration of a string array which allows you to enter a set of names into an array. It has been set to take only ten names, because you'll wear your typing fingers out if you use large arrays for examples. There are no real surprises about this program – it prints the numbers one by one, and

```

100 DIM N$(20)(10)
110 FOR N=1 TO 10
120   PRINT "Name ";N;" is ";
130   INPUT N$(N)
140 NEXT N
150 CLS
160 PRINT
170 PRINT TAB 15;"YOUR NAMES"
180 FOR N=1 TO 10
190   PRINT TAB 2;N;". "; TAB 8;N$(N)
200 NEXT N

```

*Fig. 7.4.* A string array in use.

```

100 CLS
110 GOSUB LABEL title
120 DIM N$(20)(10)
130 FOR J=1 TO 10
140   INPUT "Name- ";N$(J)
150 NEXT J
160 CLS
170 PRINT
180 PRINT "Please type initial letter ";
190 LET F=0
200 INPUT L$
210 FOR J=1 TO 10
220   IF L$=LEFT$(N$(J),1) THEN GOSUB
LABEL print
230 NEXT J
240 IF F=0 THEN PRINT "No names start
with ";L$
250 ELSE PRINT "The names above start
with ";L$
252 PRINT "Want to select another letter
(Y/N)?"
255 IF GET$="Y" THEN GOTO 160
260 END
270 LABEL print
280 PRINT N$(J)
290 LET F=1
300 RETURN
310 LABEL title
320 PRINT TAB 14;"FIND A NAME"
330 RETURN

```

*Fig. 7.5.* Finding names in a string array.

then enters the names that you type into the array. To prove that it really stores a string array, it then prints the lot.

Now that you have a string array, what can you do with it? The answer is to look for names, to group them into classes, to sort into alphabetical order, and all the other things that computers are so useful for.



Take a look at the program in Fig. 7.5. It lets you enter a string array of names, dimensioned to ten for the same merciful reasons as before. When the names have been entered, you can then ask the Lynx to print out all the names that start with a given letter. If there are no names starting with that letter, you are informed of this. All this is done by making use of a flag variable F and with a bit of string slicing. The program starts by accepting a list of names. As it stands, you have to type these in – in ‘real life’, however, they would be recorded on a tape or disk file and loaded in. You can then choose any initial letter. In line 190, the flag variable F is set to 0. Its purpose is to check whether or not any names have been found, so that the program will print the correct type of report at the end of the program.

The loop that starts in line 210 then compares the first letter of each name with the letter, allocated to L\$, that you have just typed. If they match, then the subroutine of line 270 prints the name, and sets the flag variable to 1. When all of the names have been checked, lines 240 and 250 print an appropriate message. If no matching names have been found, the value of F will still be 0, and the message in line 240 is printed. If one or more names have been found, then F=1, message in line 250 is the one that is printed. Easy, really.

This gives a hint of what can be done with string arrays, but we can't really do much more at the moment. The reason is that it gets tedious typing lists of names into the computer. Most computers provide for data filing, which means that information in the form of variables and arrays can be recorded on tape. The Lynx does not provide for this in its BASIC at present. The reason is that the manufacturers expect to have a fast-operating DATABASE program prepared to handle all the tedious programming of these actions for you. This will include all the possible selection and sorting actions, along with the recording and replaying of data. This will, of course, be essential for the business and scientific user who will want to use the Lynx with disks as storage media. Scientific and engineering users would also welcome a good set of matrix manipulation instructions.

## Chapter Eight

# Beginning Graphics and Sound

Graphics, as far as a computer is concerned, means shapes and patterns which are not the normal letters and numbers that we use for text. Modern computers provide for three types of graphics, described as *low resolution*, *high resolution*, and *user-defined*.

The resolution of graphics refers to the size of the units from which the patterns are built up. If you imagine yourself trying to create a picture with thousands of squares of coloured paper, you'll soon understand the idea of resolution. Low resolution means that your coloured squares are fairly large, so that you can't put any fine detail into your pictures. High resolution means that your coloured squares are very small, making it much easier to create pictures that show detail and can display thin lines. User-defined graphics means that you don't have to use squares – you can decide for yourself what shapes you would like to use. What we have to do now is to look at how the Lynx provides for these different types of graphics.

### The art of coarse graphics

The low resolution graphics of the Lynx consists of a choice of twenty-six characters that are permanently stored in the memory. Of these, seven are used for printing the Lynx word and the pawmark, so that your choice is really confined to the remaining nineteen. These shapes can be placed on the screen in two ways – by using the `PRINT CHR$(N)` or the `VDU` instructions, or by the use of the keyboard in its 'graphics mode'. Since the second method is less familiar we'll look at it first. When we switch on the Lynx, it is set to 'text mode', meaning that the keys will provide the usual letter and number symbols that are marked on them. The other set of characters can be switched by using the `CRTL` key and the `I` key together. The procedure is first to make sure that the shift lock is set



























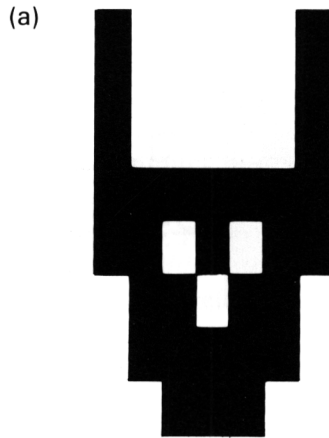
	224 f		225 a		226 b
	227 c		228 d		229 e
	230 f		231 g		232 h
	233 i		234 j		235 k
	236 l		237 m		238 n
	239 o		240 p		241 q
	242 r		243 s		244 t
	245 u		246 v		247 w
	248 x		249 y		

Fig. 8.1. The graphics symbols, their code numbers, and the keys which produce the same effects in graphics mode (after CTRL 1).

to type letters in upper-case, then press CTRL and I together. Then press RETURN. When you now press any key *and* the SHIFT key, you will get the alternative symbol for that key. For a number of

keys, this will be a graphics symbol, as the table in Fig. 8.1 shows. If you do not press the SHIFT key, you will simply get the lower-case letter for each key, which enables us to mix text and graphics anywhere on the screen. Note that you will get characters from the keys which are not alphabetical and without using SHIFT, including the £ sign from the @ key.



(b) 100 CLS  
 110 PRINT  
 120 FOR N = 1 TO 4  
 130 READ A\$  
 140 PRINTTAB17;A\$  
 150 NEXT N  
 160 DATA CTRL I J @ @ J CTRL I  
 170 DATA CTRL I J D H J CTRL I  
 180 DATA CTRL I B M I G CTRL I  
 190 DATA CTRL I @ K O A CTRL I

*Note:* In the DATA lines, press the keys as indicated – CTRL I means press both CTRL and I keys together. Do *not* use spaces – the spacing shown has been typed in to separate the letters and make them more visible. Remember that the SHIFT key has to be held down while typing the four letters in each DATA line.

*Fig. 8.2.* Putting a shape (a) on the screen, using DATA lines. This is shown as it should be typed in (b) because a listing of this program does *not* show what keys have been pressed.

Because of the control that you have over anything that is typed directly on the screen, it is easy to compose ‘pictures’ on the screen by making use of these patterns. Figure 8.2(a) shows an example of a pattern printed on the screen in this way. The method that has to be

used is slightly indirect, because of the way in which we use graphics mode. We cannot type `LET A$=""` and then place graphics characters into the string, because when we try to leave graphics mode the action of the RETURN key gives a 'syntax error'. We can, however, use `READ...DATA`, with one line of data for each line of graphics symbols, as Fig. 8.2(b) shows. Note that the letters which are shown in this listing are *not* seen on the screen – they are the letter (and other character) keys that you use in graphics mode, and each one places a graphics shape on the screen. When you LIST the program, the graphics shapes do not appear in the listing; instead you will find BASIC instruction words and spaces. This is because the codes that are used for graphics characters in graphics mode are also used for BASIC reserved words in text mode!

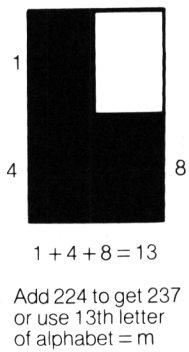
The snag with typing graphics characters direct from the keyboard, apart from the very slow printing, is that the RETURN key places a dash on the left-hand side of each character. Since the characters are in a DATA line, however, it is easy to skip reading this first character. As a method of seeing what a pattern will look like on the screen, it is reasonably quick and easy. Unfortunately, the DEL key does not carry out its normal action when the computer is in graphics mode.

1	2
4	8

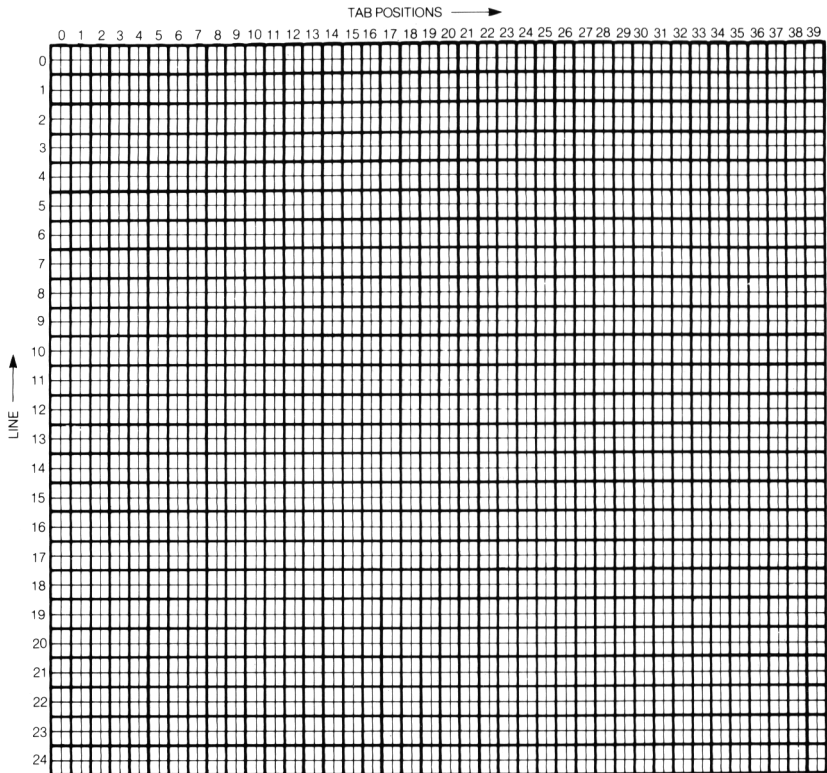
Fig. 8.3. The 'character block' which is used for the low resolution patterns.

We can also make use of these low resolution graphics patterns in programs by means of the `CHR$` and `VDU` instructions. The numbers that are listed in Fig. 8.1 show the `VDU` and `CHR$` numbers for each pattern, but a more useful approach is to see how the first sixteen patterns are built up. Figure 8.3 shows the shape of a 'character block' which is divided into four sections. The first sixteen of the graphics shapes consist of this character block with the four sections differently shaded. If you want any particular shape, then shade in the sections, and add up the numbers in the shaded sections. Add this number in turn to 224, and you have the correct `CHR$` or

VDU number for that pattern. Figure 8.4 shows an example. To design a picture using these low resolution graphics patterns, we have to start with a set of blank blocks. Figure 8.5 shows a screen diagram in which each character position is shown as having four sections. By outlining your pattern in pencil on tracing paper laid



*Fig. 8.4.* Deriving the code number for any of the 4-section blocks.



*Fig. 8.5.* A low resolution grid for planning block graphics.

over this grid, you can shade in the blocks to form a picture. You will have to decide for yourself, whether a shaded block represents white on the screen or black – you really have to be able to work either way round. You then have to write all the character numbers for the blocks that make up the pattern, and then you have to write a program to place them on the screen. One method would be to print each block individually, but this is very slow. A better method is to pack strings with the characters, using both the graphics characters and the control codes that we dealt with in Chapter 5. When you do this, remember that any string of more than 16 characters has to be dimensioned, and the limit is 127 characters. Figure 8.6 shows an example of this in action. It's an advantage to keep your pattern simple – you just can't show fine detail with low resolution graphics, and you will find that large pictures take a long time to draw. Read on, however, for a way of speeding things up!

```

100 CLS
105 DIM A$(40)
110 LET A$=""
120 FOR N=1 TO 36
130   READ J
140   LET A$=A$+CHR$(J)
150 NEXT N
160 CLS
170 PRINT
180 PRINT TAB 10;A$
190 END
200 DATA 234,224,224,234,10,8,8,8,8,
210 DATA 234,236,236,238,10,8,8,8,8,
220 DATA 226,237,233,231,10,8,8,8,8,
230 DATA 224,235,239,225,10,8,8,8,8,

```

*Fig. 8.6.* A program which draws a pattern by packing characters into a string.

## Colour and speed

The Lynx, like most modern computers, can display colour both as background (screen overall colour) and as foreground (text and pattern colour). The range of colours and their number codes is shown in Fig. 8.7. The colours are created by using three separate sets of memory to hold signals, one each for red, blue and green. These three colours are the primary colours for light; they are not the same as the primary colours for paints. The other colours that the Lynx can produce are mixtures of these three, so that magenta is a mixture of blue and red, cyan is a mixture of green and blue, and

<i>Number</i>	<i>Colour</i>
0	Black
1	Blue
2	Red
3	Magenta
4	Green
5	Cyan
6	Yellow
7	White

*Fig. 8.7.* Colours and their number codes. Either the name or the number can be used.

yellow is a mixture of green and red. White, as you probably know, is a mixture of all three primary colours.

The Lynx is unusually capable as far as colour displays are concerned, with all of its colours capable of being displayed on the screen in the same picture. With some effort, you can even produce all the colours in quite small areas, though the effect is rather wasted when you use an ordinary domestic TV for a display, as it can't do full justice to the Lynx's colour display.

## **PAPER and INK**

The Lynx follows the scheme, now used by several computers, of having the instruction words **PAPER** and **INK**. **PAPER** is used to mean the background colour, and **INK** means the foreground colour of text or patterns. When you use either of these instructions, which can also be used as direct commands, you can follow the instruction word with either the name of the colour or its number. For example, if you want to have a green background you can type **PAPER 4** or **PAPER GREEN**. If you type a line like **PRINT GREEN**, then what is printed is the number 4.

When you type **PAPER GREEN** or **PAPER 4** and press **RETURN**, the effect is not particularly noticeable, because only the background of the cursor and the **READY** that follows it will change colour. If you want the whole screen to change colour right away, you will have to follow the **PAPER** instruction with a **CLS**. Figure 8.8 shows a short program that illustrates this point. The **FOR...NEXT** loop sets up the run through the complete range



```

100 GOSUB LABEL clear
110 FOR J=0 TO 7
120   PAPER J
130   GOSUB LABEL words
140   PAUSE 10000
150   GOSUB LABEL clear
160   GOSUB LABEL words
170   PAUSE 20000
180 NEXT J
190 PAPER BLACK
200 PRINT "ALL DONE"
210 END
300 LABEL clear
310 CLS
320 PRINT
330 RETURN
340 LABEL words
350 PRINT "This is PAPER ";J
360 RETURN

```

*Fig. 8.8.* A program which illustrates the use of PAPER colours.

of PAPER colours, and the PAUSE in line 140 gives you time to see what the effect looks like before the screen is cleared. Some of the background colours will cause the text on the screen to disappear, but the cursor should be visible for most of the time. You may have to type PAPER BLACK then INK WHITE to restore things to normal after the program has run.

```

100 PAPER 7
110 CLS
120 FOR N=0 TO 7
130   INK N
140   VDU 243,244,245,246,247,248,249
150   PAUSE 10000
160   PRINT
170 NEXT N
180 PAPER BLACK
190 END

```

*Fig. 8.9.* Using different INK colours. Not all are equally useful. This is a good opportunity to check the tuning of the colour TV.

Now try Fig. 8.9 to show the appearance of text and graphics on the screen in various INK colours with one PAPER colour. Obviously, when the INK is of the same colour as the PAPER, the effect is not visible. This demonstration also shows that some INK colours are more useful and visible than others, and also that the Lynx has about the best colour display that you can get in its price range. This is also a good time to check the tuning of your colour TV – set it up so that there is the least possible flickering on INK colours,

and also set the brightness and colour controls so that the colours look correct to you.

### Colour protection

A unique feature of the Lynx colour system is the ability to 'protect' one or more of the primary colours. Protection means that anything that is on the screen in that primary colour cannot be cleared, and that the protected colour cannot be used for any other purpose. Suppose, for example, that you make the entire background of the screen green, and you then protect green. If you then put text on the screen using INK RED, the text will not appear red. This is because the red is not replacing the green – the colour that you actually see will depend on how the TV receiver is tuned. In addition, if you attempt to use a colour which is a mixture of green and any other colour (such as cyan), then only the unprotected colour will be visible. Remember in this respect that white is a mixture of all three primary colours – if you protect white, you can't produce anything! Figure 8.10 shows an illustration of colour protection using the PROTECT instruction. Note that PROTECT Ø or PROTECT BLACK removes the protection from all colours.

```

100 PAPER 4
110 CLS
120 PROTECT 4
130 PRINT
140 INK 7
150 PRINT "THIS SHOULD BE WHITE"
160 INK 1
170 PRINT
180 PRINT "THIS SHOULD BE BLUE"
190 INK 2
200 PRINT
210 PRINT "THIS SHOULD BE RED"
220 PROTECT Ø

```

*Fig. 8.10.* Making use of PROTECT – a special feature of the Lynx.

PROTECT can also be used on colours that are mixtures, like yellow, cyan or magenta. When you protect such a mixture, you are protecting both of the primary colours that it is composed of, so that PROTECT YELLOW will, for example, prevent you from using the colours RED and GREEN – we have already noted that PROTECT WHITE prevents you from using any colours until you use PROTECT BLACK. This is one way of hiding an entry so that it cannot be seen on the screen.

A side-effect of using PROTECT is that the speed of any PRINT operation is considerably increased when a colour is protected. If you want to use a program that needs fast printing on the screen (not something that the Lynx is particularly good at), then protecting two of the primary colours is one way out of the problem. If you use (this sequence is the equivalent of a TEXT command):

```
PAPER BLACK  
CLS  
PROTECT MAGENTA  
INK GREEN
```

and then carry out listing or printing you will find that these operations are carried out much more quickly than is possible when no colours are protected. If you are using a black/white receiver it will pay to carry out this routine before programming, as it speeds up listing to a much more acceptable rate.

### **Opening a window**

The Lynx does not only permit the mixture of text and graphics on the screen, it also permits mixtures of text and text! Normally when you print on the screen in text or graphics, you will also overwrite your text after the screen is full, or you will clear the screen and start again. Lynx offers another option, WINDOW. WINDOW reserves an area of the screen which can be used for text or graphics, while anything that has already been printed on the rest of the screen is left alone. It's a form of PROTECT for text and graphics and it can be very useful if you want, for example, to leave a set of instructions on the screen while you are entering data. Another possibility is to have part of the screen reserved for text and another part for drawings.

The numbers that you have to use for the WINDOW instruction are not quite so simple and straightforward, however. There are four numbers following WINDOW which specify, in order, the left-hand side position, the right-hand side position, the top and the bottom of the window. The range from left to right uses numbers from 3 to 123, and the range from top to bottom uses numbers from 5 to 245, and the whole screen will be used if the command or instruction:

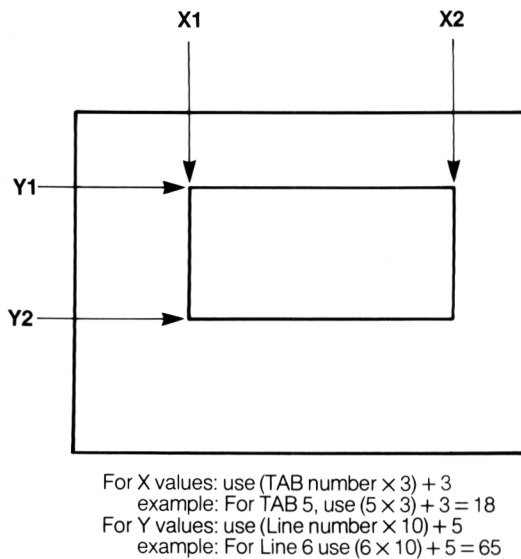
```
WINDOW 3,123,5,245
```

is used. This will be necessary if you have used a smaller window, because the normal screen area is not restored until this is done, or

until the Lynx is switched off and on again.

The numbers closely follow the numbers that we have to use with `PRINT@`. For each `TAB` position along a row, we have to multiply by three, and for each line down we have to multiply by ten. We then have to add on the starting figures of 3 for a side and 5 for the top.

For example, suppose that you have a title and three lines of printing on the top of the screen and you want these lines to remain while the lower part of the screen is used. The instruction that you need to achieve this is `WINDOW 3,123,45,245`. We've taken the X dimensions of 3 and 123 to cover the whole width of the screen, and the Y dimensions of 45 and 245 to limit the height. Since we want to protect the material in the first four lines, we take  $4 \times 10 = 40$  units to add on to the starting size of 5, so that the top number is 45. The bottom of the screen is unchanged at 245. Figure 8.11 shows how



*Fig. 8.11.* Calculating values for the `WINDOW` instruction.

`WINDOW` numbers are calculated, and Fig. 8.12 illustrates the use of `WINDOW` in a program. Note the very useful instruction `VDU23` (or `PRINT CHR$(23)`) which will place the cursor for text at the upper left-hand corner of the window space.

One unfortunate thing about the use of a `WINDOW` is that the `CLS` instruction will wipe the entire screen, not just the window. By way of compensation, though, `PRINT@` will operate even in the protected area, so that if we want to wipe the non-protected (window) space, we can use either of two methods. One is a loop

```

100 CLS
110 PRINT
120 PRINT "THIS EXAMPLE USES A WINDOW
IN WHICH"
130 PRINT "THE LISTING WILL APPEAR WHEN
THE"
140 PRINT "PROGRAM LISTS - JUST WAIT AND
WATCH!"
150 WINDOW 3,123,75,245
160 VDU 23
170 SPEED 100
175 LIST
180 REM TRY LISTING AGAIN
190 REM RESET WINDOW WITH WINDOW3,123,
5,245
200 REM RESET SPEED WITH SPEED0

```

Fig. 8.12. Illustrating the use and appearance of WINDOW in a program.

```

100 CLS
110 PRINT TAB 10;"CLEARING THE WINDOW"
120 PRINT "CLS WILL CLEAR THE WHOLE
SCREEN."
130 WINDOW 3,123,85,245
140 VDU 23
150 PRINT "THIS TEXT APPEAR IN THE
WINDOW"
160 PRINT "IT CAN BE CLEARED BY A SUB
ROUTINE"
170 PAUSE 30000
180 GOSUB LABEL clr
190 VDU 23
200 PRINT "...SEE?"
210 GOSUB LABEL fast
212 VDU 23
214 PRINT "...Better ?"
215 END
220 LABEL clr
230 VDU 23
240 FOR N=1 TO 17
250   VDU 30,31
260 NEXT N
270 RETURN
300 LABEL fast
310 PRINT "...BUT THIS IS FASTER!"
320 CLS
330 PRINT @ 6,20;"REPLACE THE TEXT WITH
PRINT@"
340 RETURN

```

Fig. 8.13. Two ways of wiping a window and leaving text on the rest of the screen.

which clears the lines of the WINDOW space, the other is a subroutine which will write the text that we want into the reserved space outside the window after a CLS has cleared the lot. Figure 8.13 shows these two ways in use. Wiping the WINDOW area, using BASIC instructions, is very slow, so that the use of the PRINT@ subroutine is always preferable.

**Low resolution sound!**

The Lynx has a sound instruction which is of a rather low resolution variety, and uses the instruction word BEEP. BEEP simply makes a sound, and you have no control over the type of sound, only over pitch, duration and volume. The sound that you hear when the Lynx is switched on is a BEEP type of note.

The BEEP instruction needs three numbers following it. The first of these numbers is described as a *wavelength number*. The wavelength number can range from 0 to 65535. Low numbers give high pitch, and high numbers correspond to low notes. Numbers at

Scale from Middle C to one octave above:

<i>Note</i>	<i>Wavelength number</i>
C	415
D	369
E	332
F	311
G	276.5
A	249
B	221
C'	207.5

For half a second of Middle C, use:

BEEP 415,1000,63

For sounds in the octave above, use half these numbers (so that E' uses  $332/2$ , which is 166).

For sounds in the octave below, multiply these numbers by 2 (so that the C below Middle C uses  $2 \times 415 = 830$ ).

*Fig. 8.14.* Numbers to use with BEEP.

the extremes of this range give sounds that you are most unlikely to hear.

The second number is a *number-of-cycles* count. The number of cycles ranges from 0 (no sound!) to 65535. The length of the note depends on this *and* on the wavelength number, multiplied together. If you use a large wavelength number and a large number of cycles, you could wait all day!

Finally, the third number controls the *volume* of the sound, ranging from 0 to 63. The notes are not particularly loud even at the highest volume setting. Volume numbers range from 0 to 63. You will generally want to use 63, because smaller numbers cause the sound to be almost inaudible except in a very quiet room. If the Lynx is connected to a hi-fi system, this volume control can be more useful.

```

100 CLS
110 LET A=415,B=500
120 FOR N=1 TO 8
130   READ J
140   LET K=A*B/J
150   BEEP J,K,63
160 NEXT N
170 DATA 415,368.8,332,311.2,276.6,249,
221.3,207.5

```

Fig. 8.15. A musical scale program.

The problem with this type of instruction is in knowing where to start, because there is no obvious relationship to musical scales. One good start is with the table in Fig. 8.14. The table shows the figures that are needed for a scale and how a half-second note at Middle C can be obtained. The wavelength number for Middle C is about 415, so that half of this number will give you a beep which is an octave higher, and double the number, 830, will give you a beep which is one octave lower. To keep the duration of each note at the same half-second, you need to arrange the number-of-cycles count so that wavelength multiplied by number of cycles is constant. Reducing the number of cycles below the amount that is obtained from this formula will give you shorter notes; increasing it will give you longer notes. You cannot, however, simply use a constant value for the number of cycles, because this will result in low notes being of much longer duration than high notes! Figure 8.15 illustrates a musical scale programmed using the BEEP instruction.

As an example of how the BEEP can be turned to good effect, Fig. 8.16 is a program that plays a snatch of melody of feline significance.

```

100 CLS
110 PRINT TAB 15;"LYNX-TROT?"
120 LET C=415,T=50
130 FOR N=1 TO 15
135   LET k=N MOD 2
140   READ J
150   LET D=C*T*(1+k)/J
160   BEEP J,D,63
170   PAUSE 100
180 NEXT N
190 END
300 DATA 207,276.6,235,249,276.6,332,311
310 DATA 369,332,415,369,442.6,415,830,
830
320 REM With deferential apologies to
Andrew Lloyd-Webber.

```

*Fig. 8.16.* Suitable music for the slim Lynx. Apologies and acknowledgements to the composer, Andrew Lloyd Webber.

```

100 CLS
110 PRINT TAB 14;"SIREN-SOUND"
120 LET C=415,T=10
130 FOR N=C TO 100 STEP -10
140   BEEP N,C*T/N,63
150 NEXT N

```

*Fig. 8.17.* Programming a note that rises in pitch.

```

100 CLS
110 PRINT TAB 17;"WARBLE"
120 PRINT TAB 9;"(To stop,press any key)"
130 REPEAT
140   BEEP 225,50,63
150   BEEP 200,50,63
160 UNTIL NOTKEY$=""

```

*Fig. 8.18.* A warbling note, always a good attention-getter.

```

100 CLS
110 PRINT TAB 16;"VOL-BEEP"
115 REPEAT
120   FOR N=1 TO 63 STEP 4
130     BEEP 100,50,N
140   NEXT N
145 UNTIL NOTKEY$=""

```

*Fig. 8.19.* Changing volume during a note.

We can also use BEEP as a way of signalling various events in a game, as a warning (DO NOT PRESS RETURN), or as a way of creating sound effects. Figure 8.17 shows how BEEP can be programmed to produce a rising note by putting the wavelength



number as a variable and having the number-of-cycles count as a constant divided by the other variable. This preserves the length of the note as the pitch changes.

A variation of this in Fig. 8.18 is a warbling note. This is an excellent attention-getter, and if the notes are fairly close in pitch and the volume high it isn't difficult to ensure attention. Another variation on the theme is the change of volume as the note sounds, and this is illustrated in Fig. 8.19.

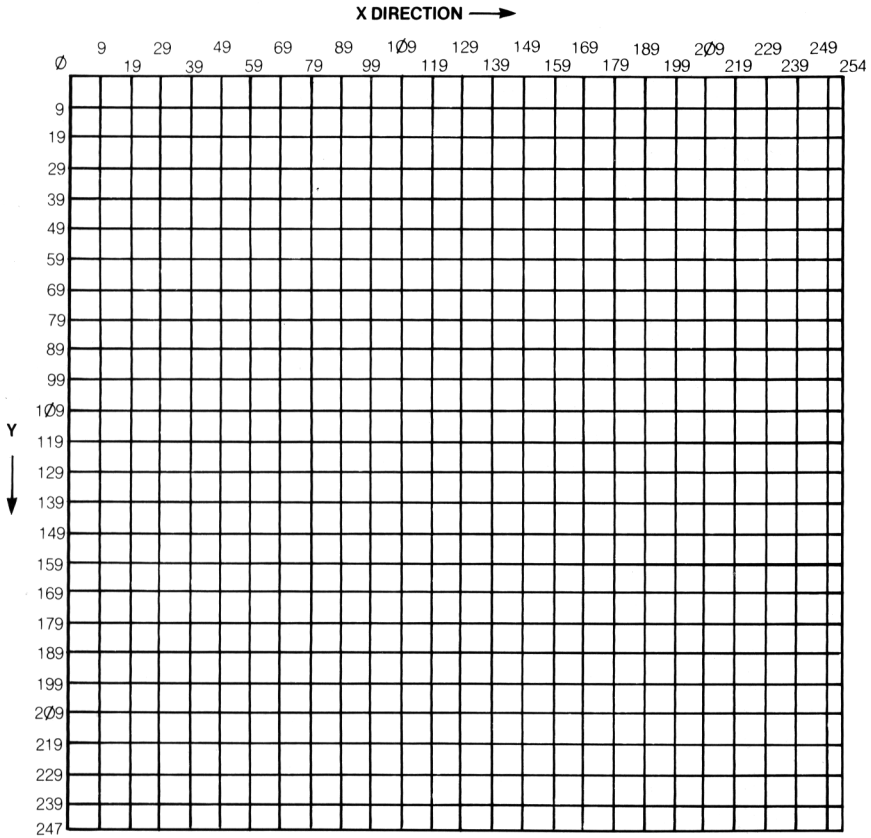
## Chapter Nine

# Higher Resolution All Round

The crowning glory of graphics for any computer is the provision of high resolution graphics. There is a lot of variation, as you look from one computer to another, about what constitutes high resolution, but there is at least some agreement about how we measure resolution. This is measured in terms of *pixels*, or *picture elements*. These are the small dots (more correctly, rectangles) that we use to make up a picture, and the number of pixels that you can place on the screen is a good measure of resolution. If we hark back to the low resolution graphics of the Lynx, we used four blocks per character position. Since there are 40 characters per line and 25 lines, that makes  $4 \times 40 \times 25$  blocks, a total of 4000 pixels. The high resolution graphics of the Lynx allows us to use 256 pixels across the screen and 248 down, which is a total of  $256 \times 248 = 62000$  pixels. That's one item of good news – the other is that we don't have to write programs that turn on each pixel in turn. The Lynx has a useful set of instructions which will allow us to draw pictures using these pixels. Before we start on picture drawing, however, we need to know something about how we can specify position on the screen.

The Lynx follows the method that is practically a standard. The 256 positions across the screen are numbered, starting from the left-hand side at 0 and going up to the right hand side at 255 (Fig. 9.1). The positions down the screen are similarly numbered 0 to 247, with the 0 position being at the top of the screen. The top left-hand corner of the screen is the position where both numbers are zero, so that we can refer to it as 0,0. These two position numbers are always put in the order of across, down (X,Y if you are used to graphs), so that position 80,60 means 80 pixels across from the left-hand side and 60 pixels down from the top. These numbers are called *co-ordinates*. When we use co-ordinates for specifying positions on the screen of the Lynx, we can use numbers, variables, or expressions.

Drawings are made on the Lynx screen by moving a cursor which



*Fig. 9.1.* The high resolution graphics planning grid for Lynx.

is normally invisible while we are using graphics. There are instructions which will move the cursor without leaving a trace, for moving the cursor with a trail in the INK colour, and for placing a dot at the cursor position. In addition, there is one very useful 'do-all' instruction which uses control codes of its own to specify what is to be done. Let's start with cursor movement and drawing.

### Move and draw

The MOVE instruction places the cursor in a new position, the position that is given by the co-ordinate numbers that follow MOVE. The cursor normally starts, after you have cleared the screen, at the 'home' position of  $\emptyset, \emptyset$ . Though the graphics cursor is invisible, the ordinary text cursor is not, and it's usually a good idea

to turn it off by using VDU15 before you start. If you then use the instruction:

```
MOVE 122,123
```

then the cursor will be placed at the centre of the screen – still invisible. This allows you to place the cursor at the start of a drawing without leaving a trail.

The next instruction is DRAW. DRAW is also followed by two co-ordinate numbers, and its effect is to move the cursor from wherever it was placed before to the new position specified by the co-ordinates, leaving a straight line trail. This trail is one pixel wide, and will be in whatever colour you have selected by the use of INK in the program. Both DRAW and MOVE use ‘absolute co-ordinates’, meaning that the numbers which follow the instruction words are screen co-ordinate positions, not distances measured from the previous position of the cursor. If you want to plan out a drawing, therefore, you must label each change of direction with the co-ordinates of the point where the lines meet.

Figure 9.2 shows an example. The shape is a very simple one, because I want to illustrate how the instructions are used rather than baffle you with artistry (who – me?). Just to illustrate how the Lynx can place a lot of colours on the screen, I have used the full range of colours for the shape.

```
100 CLS
110 VDU 15
120 MOVE 84,80
130 INK BLUE
140 DRAW 84,200
150 INK GREEN
160 DRAW 172,200
170 INK RED
180 DRAW 172,80
190 INK YELLOW
200 DRAW 152,80
210 INK CYAN
220 DRAW 152,120
230 INK MAGENTA
240 DRAW 104,120
250 INK WHITE
260 DRAW 104,80
270 INK GREEN
280 DRAW 84,80
290 END
```

*Fig. 9.2. Drawing a shape, using straight lines.*

How do we go about drawing a pattern on the screen? The answer is, as usual, with a lot of planning and hard work. We start by placing a sheet of tracing paper over the  $256 \times 248$  grid pattern of Fig. 9.1. If you have only small size tracing paper, don't be too worried because you generally use a lot less than the maximum size of the screen for any drawing. Clip the tracing paper securely over the screen diagram. If the diagram reproduced here is too small for you, make up your own by ruling off a piece of A3 graph paper which is scaled in mm, 5 mm and cm divisions. Number the cm divisions as 0, 9, 19, 29.... positions from the home corner, and you are ready to go!

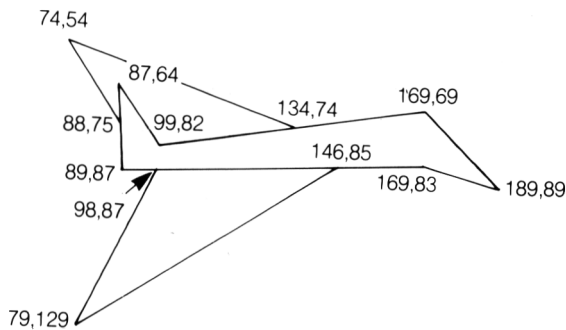


Fig. 9.3. A shape traced out, showing co-ordinates.

Sketch roughly the outlines that you want (Fig. 9.3). Go for broad outlines rather than fine detail, because fine detail of one pixel or so may not be seen, particularly on a colour TV screen. Don't worry about sweeping curves, because you can produce a reasonable approximation to a curve with a set of straight lines. A few computers have instructions for creating portions of circles, but when you examine these circles, you find that they also are made up from straight lines – the only advantage of a **CIRCLE** instruction is easier programming. See Appendix F for a suitable procedure.

The next step is to place a dot wherever two straight lines meet. You will then have to write down the co-ordinate numbers for each of the dots, looking up to the top of the grid to find the X numbers, and to the left-hand side to read the Y number. Once you have a pair of co-ordinate numbers for each point, you are ready to program your drawing.

The program for this example is shown in Fig. 9.4. So as to avoid a lot of **DRAW** instructions, **FOR...NEXT** loops have been used to place the co-ordinate numbers into the **DRAW** instructions. The reason for breaking the program into sections is to allow the **MOVE**

```

100 CLS
110 VDU 15
120 MOVE 189,89
130 FOR N=1 TO 7
140   READ a,b
150   DRAW a,b
160 NEXT N
170 MOVE 88,70
180 FOR N=1 TO 4
190   READ a,b
200   DRAW a,b
210 NEXT N
220 MOVE 88,87
230 FOR N=1 TO 3
240   READ a,b
250   DRAW a,b
260 NEXT N
270 DATA 169,69,134,74,74,54,88,70,87,
64,99,82,134,74
280 DATA 89,87,98,82,79,129,146,85,146,
85,169,83,189,89

```

*Fig. 9.4.* A program to produce the shape on the screen.

instruction to shift the cursor so that we don't trace over a previously drawn section. We've made no attempt in this example to use colour – one thing at a time!

The position of this drawing on the screen is absolute. Absolute means that if we want to change the position of the drawing on the screen we will have to change each of the co-ordinate numbers. That's hard work, and the Lynx provides for a different type of instruction, as we'll see later, which permits movement simply by changing one pair of co-ordinates. This is done by making every DRAW instruction a relative one.

Before we leave the DRAW and MOVE instructions, however, take a look at Fig. 9.5 which draws an appropriate shape. One of the charms of high resolution graphics is that it's very difficult for anyone reading the listing to find out easily what the shape will be! In this example, the main shape is drawn using the FOR...NEXT loop in lines 150 to 180 and with the X and Y co-ordinate numbers in the DATA lines 360 to 410. The embellishments are put in using lines 190 to 350, with a splash of colour here and there.

## **DOT and graphs**

The DOT instruction is a particularly useful one for plotting graphs, and is another example of the thought that has gone into making the

```

100 PAPER BLACK
110 INK YELLOW
120 CLS
130 VDU 15
140 MOVE 252,157
150 FOR N=1 TO 19
160   READ a,b
170   DRAW a,b
180 NEXT N
190 MOVE 201,67
200 DRAW 202,65
210 DRAW 207,63
220 MOVE 175,67
230 DRAW 165,63
240 DRAW 165,63
245 INK WHITE
250 MOVE 219,45
260 DRAW 239,28
270 MOVE 215,39
280 DRAW 229,22
290 MOVE 169,39
300 DRAW 147,24
310 MOVE 167,45
320 DRAW 139,32
325 INK RED
330 MOVE 179,79
340 DRAW 192,83
350 DRAW 203,79
360 DATA 225,136,248,134,235,122
370 DATA 219,75,229,59,219,29
380 DATA 209,49,189,45,170,49
390 DATA 169,29,149,61,162,76
400 DATA 143,95,109,88,80,86
410 DATA 47,70,47,144,205,149,252,157

```

Fig. 9.5. Another shape-tracing program – can you guess the shape?

Lynx so suitable for the scientific, engineering and business user as well as for home computing purposes. The DOT instruction needs to have two co-ordinate numbers following it. As usual, these can be numbers, number variables, or expressions. The effect, as you might guess, is to print a dot in the INK colour that is being used at the co-ordinate position that is specified. Every single dot on the screen can have its colour controlled individually. This contrasts well with many computers which will permit a dot only to be in the same colour as the group of dots around it.

The most useful feature of DOT is the ability that it gives you to draw graphs of complicated shapes, as Fig. 9.6 shows. This example illustrates an educational application of Lynx – the ability to draw a graph that shows a complex waveshape being drawn, using a set of

```

100 CLS
110 VDU 15
120 MOVE 0,120
130 FOR N=1 TO 500 STEP 2
140   LET Y=120-50*(SIN(RAD(N))-0.5*SIN
(RAD(2*N))+0.3*SIN(RAD(3*N)))
150   DOT N/2,Y
160 NEXT N
170 PRINT @ 3,200;"Third harmonic dist
ortion!"
180 LET A$=GET$

```

*Fig. 9.6.* Drawing a graph of a mixture of waves, using DOT.

sine wave equations. For anyone who is grappling with Fourier series, this could be a gift!

### A new plot

The disadvantage of using the MOVE, DRAW and DOT instructions, as we have seen, is that they need absolute co-ordinates. Another method of creating drawings makes use of *relative* co-ordinates. Drawing in relative co-ordinates means that you fix a starting point, using MOVE with two co-ordinates, but follow this with instructions that use relative co-ordinates. This means that every position is plotted relative to the previous position. When you say 'five steps left and two forward', you are dealing in relative co-ordinates. Where you will end up as a result of such an instruction depends on where you were when you started. If you place the graphics cursor at the centre of the screen by using MOVE127,124, then you can use relative co-ordinates of -10,0 to mean that your next position will be 117,124. That's ten subtracted from the 127 co-ordinate and nothing from the Y co-ordinate. We might equally well have used 0,20 to mean a shift to 127,144 – no change in the X co-ordinate, but an increase of 20 in the Y co-ordinate.

Lynx uses an instruction, PLOT, which allows you to use MOVE, DRAW and DOT in one instruction. In addition, it permits you to use MOVE and DRAW with relative co-ordinates, so making it possible to create a pattern which can be shifted to different places on the screen. The key to this is the use of an additional number, the *mode number*, which is the first number in the PLOT instruction. The mode numbers for PLOT are summarised in Fig. 9.7.



Mode Number	Effect
0	Action of MOVE (absolute co-ordinates)
1	Action of MOVE (relative co-ordinates)
2	Action of DRAW (absolute co-ordinates)
3	Action of DRAW (relative co-ordinates)
4	Action of DOT (absolute co-ordinates)

Fig. 9.7. The PLOT mode numbers which have to be followed by the usual X, Y co-ordinates. This offers more flexibility in drawing.

```

90 PAPER BLACK
100 CLS
110 VDU 15
120 MOVE 50,80
130 FOR N=1 TO 40
140   INK YELLOW
150   GOSUB LABEL draw
170   INK BLACK
180   GOSUB LABEL draw
190   PLOT 1,5,0
200 NEXT N
210 END
220 LABEL draw
230 PLOT 3,-50,-30
240 PLOT 3,30,30
250 PLOT 3,-30,30
260 PLOT 3,50,-30
270 RETURN

```

Fig. 9.8. An animated drawing, using PLOT both to draw and to erase.

As an illustration of the way in which you can use PLOT to draw with relative co-ordinates, Fig. 9.8 draws a simple shape in 40 different positions across the screen. Each drawing is done using a subroutine, and by carrying out one drawing in yellow and the other in black, the effect of the subroutine is to create a drawing in yellow and then to erase it. This is because black is the PAPER colour, and black lines on black paper make the drawing invisible. If we had used blue PAPER, we would have to have made the second drawing in blue. After the drawing has been erased in this way, the cursor is moved, and the drawing cycle repeats.

It's possible to animate a drawing by successively drawing, shifting, drawing, shifting back, erasing, and so on, but the BASIC language of the Lynx is very poorly suited to such processes. BASIC simply

isn't fast enough, and the rather slow screen display actions (PRINT or DRAW) make animation from a BASIC program rather unlife-like unless you are illustrating the life-cycle of the three-toed sloth. The sort of animations that you see in arcade games have to be tackled by different methods, writing the program directly to control the microprocessor that is the heart of the computer. That's a big task, and a text on machine code would fill a book twice the size of this one, so we'll leave it out. There is an alternative language, called FORTH, which permits much faster action, and when this becomes available on the Lynx, programmers will better be able to utilise the many excellent graphics features of the machine. For the meantime, however, you can try the effect of INK GREEN and PROTECT MAGENTA to speed things up, and Fig. 9.9 shows the improvement that drawing one image before deleting the previous one can make.

```

90 PAPER BLACK
90 PAPER BLACK
100 CLS
110 VDU 15
115 PROTECT MAGENTA
120 MOVE 50,80
130 FOR N=1 TO 40
140   INK GREEN
150   GOSUB LABEL draw
152   PLOT 1,5,0
155   GOSUB LABEL draw
157   PLOT 1,-5,0
170   INK BLACK
180   GOSUB LABEL draw
190   PLOT 1,5,0
200 NEXT N
210 END
220 LABEL draw
230 PLOT 3,-50,-30
240 PLOT 3,30,30
250 PLOT 3,-30,30
260 PLOT 3,50,-30
270 RETURN

```

*Fig. 9.9.* Improving animation by printing one place along before deleting the old drawing.

### **Do-it-yourself characters**

Every now and again you may feel the need to use Greek characters on your Lynx. This is particularly likely if you are working with

scientific or engineering equations – or just learning Greek! You may, perhaps, also like to design your own space-ship, alien figure, or other shapes to order. For shapes that are too small for the DRAW instruction, Lynx offers you the opportunity to create a number of ‘user-defined’ characters. The code numbers 128 to 255 (a total of 128 characters) can all be used for these purposes. Normally, if you use VDU193 or PRINT CHR\$(193), you will get the letter A, just as if you had used the code 65. The piece of memory which stores these codes is filled with a copy of the ordinary set of characters when the Lynx is switched on.

This piece of memory, however, is one that can be changed, meaning that we can store different codes there. In particular, we can store codes which will provide the shapes for the characters that we want, in place of the codes for the ordinary characters. This process requires two new instructions to be explained, PEEK and POKE.

PEEK means ‘find what is stored in memory’. The memory of a computer consists of units called *bytes*, and each byte has its own reference number, called the *address*. One byte can store a number which is any value between 0 and 255, and the purpose of PEEK is to discover what is stored at any address number. Now with the 65536 different addresses that are available in the Lynx, you can’t just go looking at random – you need to know where to look for the graphics characters. Lynx makes this easy for you by using the word GRAPHIC. Typing PRINT GRAPHIC gives an address number – 25201 when I tried it on mine. This is *not* where the characters are stored, though, only a ‘pointer’ to the characters. A pointer is an address which holds another address – just as you can go to the address of a tourist office to find the address of a hotel. A pointer is a sort of tourist office for an address, and two pointer addresses are needed to hold any one address number. The reason is that a byte can store only a number up to 255. What is done is to split an address up into units of 256. The first pointer address holds the remainder of the division (address MOD 256) and the second pointer address holds the integer part of the division (address DIV 256). To see this in action, take a look at the numbers that are stored in the GRAPHIC address of 25201 and the next one up, 25202. When we type PRINT PEEK (25201) we get 212 on pressing the RETURN key, so this is the number that is stored at this address. PRINT PEEK(25202) gives the number 1. Now the higher pointer address of 25202 gives the higher part of the address that we want, and the 1 in this position means  $1 \times 256$ . The whole address, then is  $1 \times 256 +$

212 = 468. This, at last, is the address that we want. What exactly do we do with it?

The graphics and other characters are stored using ten bytes of memory each. Suppose we take a look at the first fifty bytes that are stored from address 468 onwards. We can do this by using the program that is illustrated in Fig. 9.10. This prints the numbers in lines of ten, one line for each character.

```
100 CLS
110 FOR J=0 TO 49 STEP 10
120   FOR K=0 TO 9
130     PRINT PEEK(468+J+K); " ";
140   NEXT K
150   PRINT
160 NEXT J
```

Fig. 9.10. PEEKing at memory so as to find the codes for characters.

What does this program give us? The first line is for the first character – and it consists entirely of zeros. That’s because the first character of the ASCII set, which is copied here, is the space, ASCII code 32 which is also used in the Lynx for code 128. Things get more interesting when we look at the second line, which reads 0,8,8,8,8,8,8,0,8,0. This, believe it or not, is the exclamation mark!

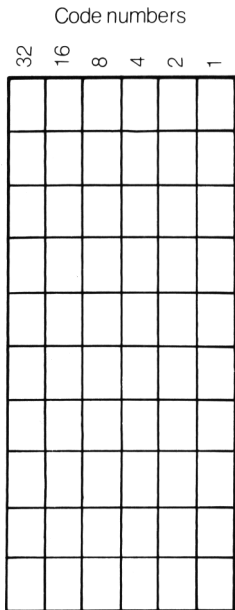


Fig. 9.11. The character-planning grid.

To see why, look at Fig. 9.11. This shows a set of 60 boxes, arranged as ten rows of 6, and with the boxes numbered. The numbers don't run 1,2,3,... but 1,2,4,8,16 and 32; each number is double the size of the number to its right. The number that we find stored in memory will correspond to numbers of these boxes. The computer is arranged so that a code number 8 stored at one of these addresses will cause a dot in the position of that box to be illuminated. If we represent this action by shading in boxes, we can see what shapes are represented by the figures that the program of Fig. 9.9 produces. Place a piece of tracing paper over the grid, and shade in the boxes. Taking the second row of figures that we got from the program,  $\emptyset, 8, 8, 8, 8, 8, \emptyset, 8, \emptyset$ , we need to shade in the number 8 box in rows 2,3,4,5,6,7 and 9 only. No other boxes are shaded. This gives you the shape of the exclamation mark. Try it out on some of the other rows of figures, and you will see how the shapes are built up. A number like 12 means that the 8-box and the 4-box are filled; 25 means that the 16-box, the 8-box and the 1-box are filled and so on. Since each character is placed on a block that is ten rows of six boxes, we could have up to 60 dots in a character. In practice, we normally leave a blank row top and bottom so that lines of characters are separated. We also leave room at one side or both so that characters do not appear to be joined together on the screen.

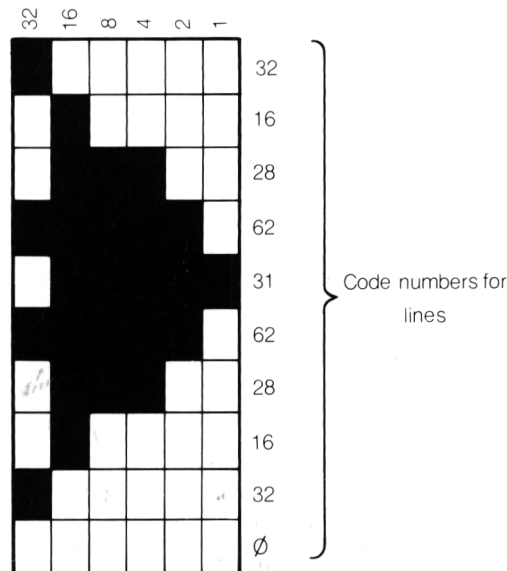


Fig. 9.12. A delta-wing shape shaded on to a character block.

This way of storing in coded form what appears on the screen also allows us to create character shapes for ourselves. Suppose, for example, that we want to create a small 'delta-wing' shape. We start with the grid of  $6 \times 10$  squares as illustrated in Fig. 9.11. On this we draw the shape that we want, using tracing paper and soft pencil. Figure 9.12 shows how we might shade in the blocks to create this shape. Remember that where we are shading, the computer will light a dot, unless we have selected **INK BLACK** and **PAPER WHITE**.

Having shaded in the boxes, we can now find a number code for each row. The 'row code number' is obtained by adding up the code numbers for the shaded boxes. In the first row, the left-hand box is shaded, so the code for that row is 32. In the second row, the box in the code 16 position is shaded, so that the code for this one is 16. In the third row the shaded boxes are in the 16, 8, and 4 positions, adding up to 28. This process of adding up the codes for the shaded boxes is continued until we have a number for each row. These numbers can be written at the side of the diagram, as we have done in Fig. 9.12.

The next step is one of programming. We have to place these numbers into the memory of the computer, and we also have to ensure that the computer stores the starting address. This is because we store the numbers in sequence, and if the computer has a record of the address at which the first number is stored, then it can find the others by counting through the numbers. There will be ten numbers for each character, so that the count is straightforward.

This, however, is where things get complicated. First of all, we have to select and protect a chunk of memory. Normally, the computer decides for itself how it will make use of its memory. As a result, unless we prevent it from using a piece of memory, we can never be sure that anything that we store there will be safe. The instruction word that we need for this task is **RESERVE**. **RESERVE** means 'rope-off memory', and the easiest part of the Lynx memory to rope off is the piece at the top end, the highest addresses that the computer uses.

Type **PRINT HIMEM**, and then press the **RETURN** key. This gives you the number 40952, which is the highest address in the memory that the machine will use. Now if you reserve ten units of this, so that the machine cannot use any address higher than 40942, then you have a piece of roped-off memory, kept safe for you until you want to change it, or until the machine is switched off. The instruction that will carry this out is:

**RESERVE HIMEM-10**

The next step in creating a character is to change pointers. At present, the GRAPHIC pointer of the computer points to the address 468. We are going to change this, because we are going to store our character codes starting at the HIMEM-10 address, so the pointer has to be altered. This, on some computers, needs two operations because there are two pointer addresses to change. The Lynx, like the good ol' Nascom, uses an instruction which will alter both of the pointer addresses in one go. The Lynx version of this instruction is DPOKE (Nascom uses DOKE), meaning Double POKE – two numbers are placed into two addresses by one instruction. DPOKE has to be followed by two numbers. The first one is the address of the first of the pair, the second number is the quantity that you need to place in these addresses. When we are designing characters we don't have to memorise the numbers, we can simply use words, so that:

DPOKE GRAPHIC,HIMEM

will do what we want. GRAPHIC refers to the pointer addresses, and HIMEM is the new value of the highest memory address, ten less than it was when we switched on because of the effect of RESERVE. When the DPOKE instruction has been carried out, the graphics pointer addresses will now contain numbers that give the address of the new HIMEM value, which is where we shall store the codes for our character. We now have to fill these ten bytes of memory with our own code bytes. The characters that are 'pointed to' by GRAPHIC are these with code numbers 128 upwards. The first byte of the character whose ASCII code is 128 will normally be stored at the address given by GRAPHIC and GRAPHIC+1. Once again, Lynx offers an easier way of carrying out this process – LETTER(128) will lead to this address.

```

100 CLS
110 VDU 15
120 RESERVE HIMEM-10
130 DPOKE GRAPHIC,HIMEM
140 FOR N=0 TO 9
150   READ J
160   POKE LETTER(128)+N,J
170 NEXT N
180 DATA 32,16,28,62,31,62,28,16,32,0
190 CLS
200 PRINT
210 PRINT TAB 10;CHR$(128)
```

*Fig. 9.13.* The program which creates the pattern and prints it.

The next part of the operation uses the **POKE** instruction. Unlike **DPOKE**, **POKE** places one number into one address at a time. **POKE** has to be followed by two numbers, the address number, and then the number (0 to 255) which you want to store at that address. The address for the first number in the example is **LETTER(128)**, and the number is 32, so that **POKE LETTER(128),32** will do the needful for this first new number.

It's rather painful to have to **POKE** ten lots, though, so we use a **FOR...NEXT** loop to do the job. We have to start with a value of 0, because we must ensure that we start with the address given by **LETTER(128)** and go up to **LETTER(128)+9** – that's a total of ten addresses. Figure 9.13 shows the results of all this action. The memory is reserved, the pointer is set, the numbers are **POKEd** into place, and finally the result is printed. This differs, you will notice, from the method that is shown in the manual which used **BIN** codes, putting a 1 in each shaded box. The advantage of the method that we have described here is that it needs less typing!

All of this programming gives us a character whose ASCII code is 128 and which is made-to-measure for our needs. It's no more trouble to create a whole set of such characters than it is to make one. We can print the characters anywhere on the screen, using **PRINT@**, and shift them from place to place. We can also combine each character in a string with other characters, including control codes, using the methods that we investigated in Chapter 5. All in all, it's a useful provision.

The size of each character is rather small, and for some purposes, it's better to use more than one to make up a shape. The method of creating each character is pretty much the same; except for two characters we have to reserve 20 units of memory rather than just ten. Once we have defined two characters, 128 and 129, we can then combine them by defining a string as these two characters. A **PRINT A\$** instruction will then place the characters together on the screen wherever we specify. The technique is demonstrated in Fig. 9.14, using a shape which I have termed a 'hexapuss'. If you prefer to think of it as an oil rig, that's up to you!

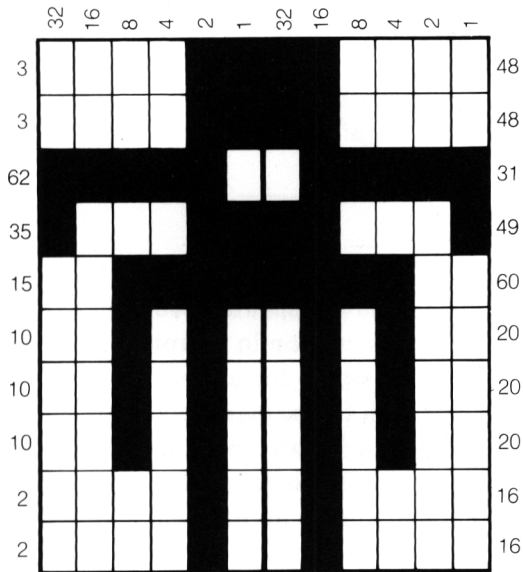
I found, incidentally, that these new characters had an unexpected side effect on my Lynx – they changed the shape of the cursor! This does no harm, but I could not change the shape back by using any of the normal methods (see **CCHAR** in Chapter 10), and had to switch off and on again to restore the normal cursor. This must be because the cursor is formed from one of the graphics shapes that has been redefined.



(i)

(a)

(b)



Codes for (a)

Codes for (b)

```

(ii) 100 CLS
      110 VDU15
      120 RESERVE HIMEM-20
      130 DPOKE GRAPHIC,HIMEM
      140 FOR N=0 TO 19
      150   READ J
      160   POKE LETTER(128)+N,J
      170 NEXT N
      180 LET A#=CHR$(128)+CHR$(129)
      190 PRINT
      200 PRINT TAB 10;A$
      300 DATA 3,3,62,35,15,10,10,10,2,2
      310 DATA 48,48,31,49,60,20,20,20,16,16
      320 FOR Y=1 TO 24
      330   PRINT @ 60,10*Y;A$
      350 PRINT @ 60,10*Y;" "
      370 PRINT CHR$(29);
      380 NEXT Y

```

Fig. 9.14. (i) The hexapuss shape and (ii) the program (well, if you can have an octopus ...).

## Chapter Ten

# More Sound and Some Afterthoughts

We introduced the BEEP instruction in Chapter 8, as a sort of ‘low resolution’ sound. The title was a bit unfair, but there is another Lynx sound instruction which can be manipulated much more freely than BEEP. It’s SOUND, and the way in which it converts numbers into sound is unlike the SOUND instruction of any other computer. Unfortunately, the manual shows no examples, so we have to go through the principles step-by-step.

The basis of the Lynx sound generator is a ‘digital-to-analog’ converter circuit. This means a circuit which will convert a set of numbers into electrical voltages, so that each voltage bears some relationship to each number. If we apply a collection of varying voltages to a loudspeaker, we will hear a sound. Simple, really! What’s less simple is being able to specify the correct numbers. To do so, we need to have some idea of what sound is and how it is produced.

Sound consists of vibrations in the air, causing the pressure of the air to change many times per second. The amount of pressure change is called the *amplitude* of the sound (Fig. 10.1), and the number of times the pressure goes through a complete cycle of changes is called the *frequency*. Our ears detect the amplitude of sound as its *loudness* (there’s more to it than that, but it’s close enough for the moment), and the frequency as its *pitch*. The frequency range that we can detect is from around 40 complete cycles per second (40 hertz) to around 15000 per second (15000 hertz).

These two quantities, amplitude and frequency, however, are not enough to specify a sound completely. You can tell the difference between a violin and a clarinet, even when they are playing the same note equally loudly. The *waveshape* of a note is the shape of the graph of pressure plotted against time (for a note carried through the air), and each musical instrument produces a different waveshape.

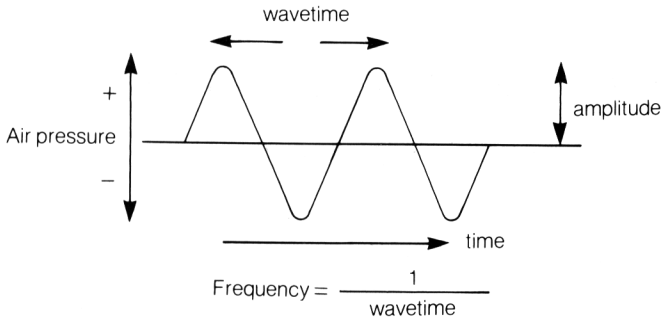


Fig. 10.1. The amplitude and frequency of a sound wave illustrated by a graph.

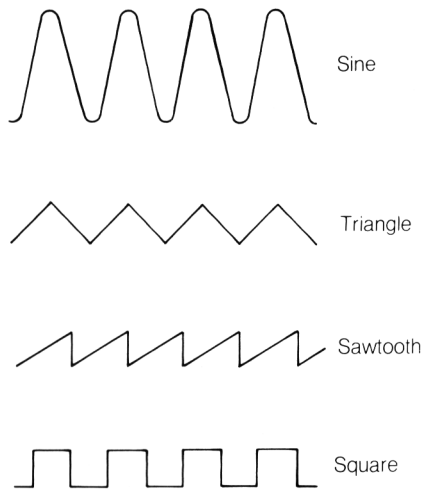


Fig. 10.2. Some of the waveshapes that are used in electronic instruments.

Figure 10.2 shows some of the waveshapes that are used in modern electronic instruments. In addition to all that, each musical note consists of a large number of waves whose amplitude changes during the time of the note. We can identify four separate sections of a note (Fig. 10.3) which make up what we call the *envelope* of the note. The first of these sections is *attack*, where the amplitude of each wave is greater than the amplitude of the one before it at the start of the note. Some instruments have a fairly slow attack; others, notably the piano, have a fast attack because the action is one of striking strings with hammers. After the attack section there may be a *decay*, as the amplitude decreases, then a *sustain* section with constant amplitude. Some instruments will feature a decay to zero amplitude, with no sustain section (piano, again); others will have a sustain, with very

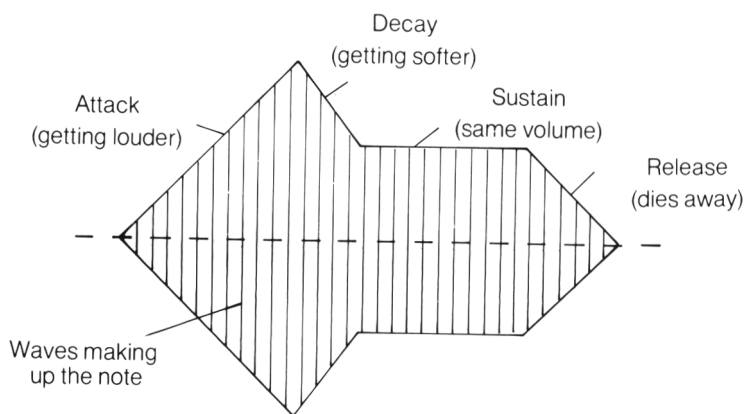


Fig. 10.3. The parts of a sound 'envelope'.

little decay (wind instruments). Finally, the note dies away in a *release* section.

To use the **SOUND** instruction of the Lynx, then, you have to write down a set of numbers that indicate the shape of the waves and also of the envelope. These numbers will have to be stored in the memory of the computer. A **SOUND** instruction will then send the numbers to the digital-to-analog converter at whatever speed you specify so that you can produce notes of whatever pitch you want. You can produce a good range of different types of notes by storing several sets of numbers that give waves of different shapes, with different envelopes, and with different maximum amplitudes. The **SOUND** instruction then allows you to pick out whichever one you want.

Three-part harmony? It's not exactly easy unless you can find the wave shape that you want. Harmony is produced when you have waves at different frequencies. It's most easily produced by having separate generators, but the signals have to be added into one single wave in order to be fed to a loudspeaker, so it's certainly possible to specify such a set of notes in one **SOUND** instruction set of numbers. The method involves drawing a wave which contains another wave – but unless you are very familiar with the waveshapes of sounds, trial and error is the only way open to you.

How do we start, then? The answer starts with the chart of Fig. 10.4. This looks like yet another graphics planner chart, but the Y direction is numbered 0 to 255, which is the maximum number range for storing in one byte of memory. The numbering on the X-axis is not so significant – the numbers on this axis are there only as reference numbers to guide you, they don't correspond to any

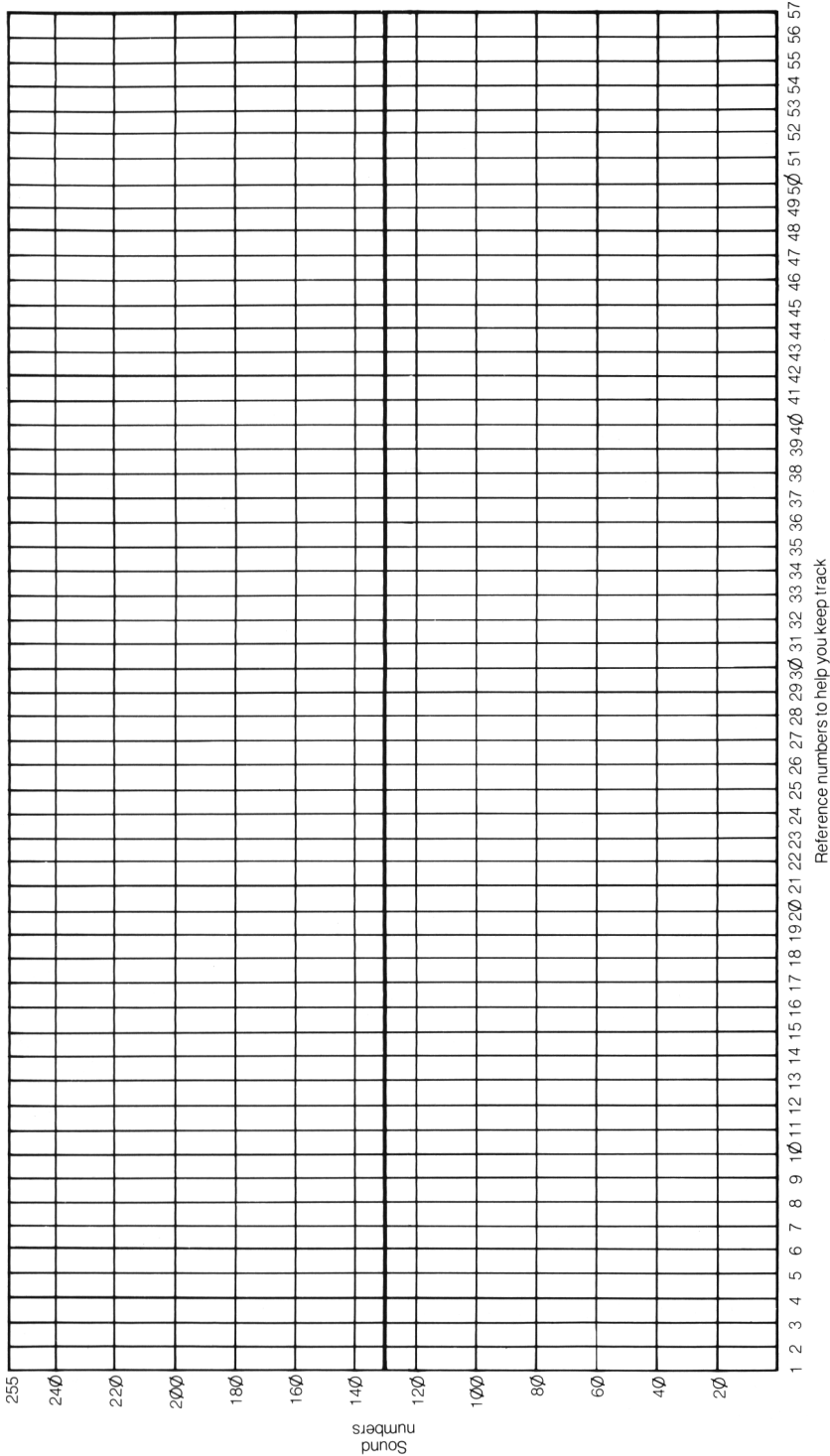


Fig. 10.4. A sound planning chart for the SOUND instruction.

quantity that you need to use. The mid-way mark of the Y-axis, at around 127, is marked in.

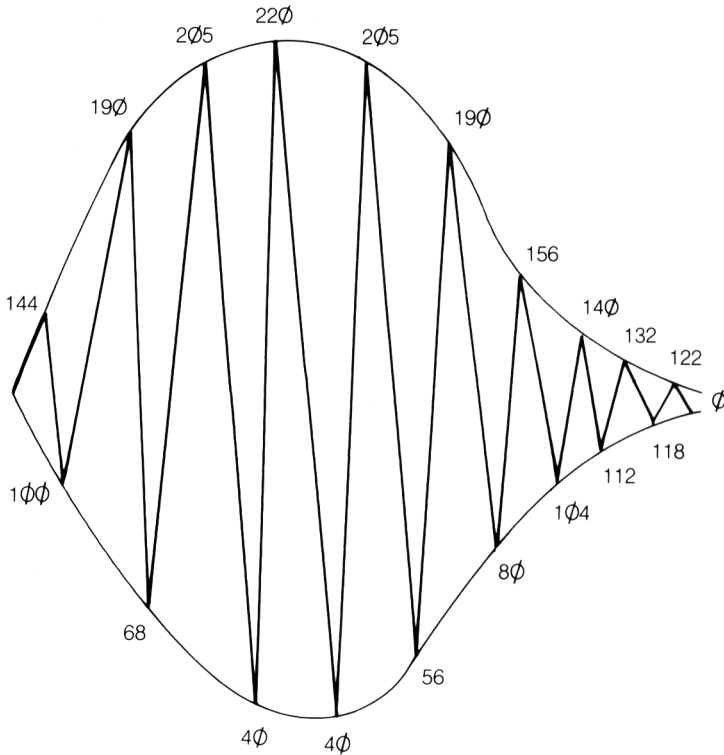


Fig. 10.5. A sound envelope to try.

On this chart, we draw the outline of the waves and their envelope. If the envelope is a very complicated one, we might want to extend the chart considerably in the X direction, but the existing scheme is good enough for what we want to try out. I've illustrated a sample of the technique in Fig. 10.5. The last number that is used is  $\emptyset$  rather than 127, because the Lynx uses the amount  $\emptyset$  to detect the end of a stream of SOUND numbers. What I have done is to draw the waves and their envelope, and then read a number from the Y axis at each change of direction of the lines. If this sequence of numbers is now stored in the memory, the computer can use them as the specification for a note each time the SOUND instruction is used. Now there are 20 numbers, including the final zero, in this example, so to store them in memory, we need to reserve space by using `RESERVE HIMEM-20`. We can then `POKE` the numbers into the memory at address HIMEM onwards, and convert them into sound by using `SOUND HIMEM,10000`. The SOUND instruction needs to

be followed by two numbers. One of these is the starting address for the sound numbers, the other is the speed number which specifies how fast the numbers are converted into voltages. We've used 1000 as a start – the range is up to 65535. As it happens, 1000 gives a low note, and values in the 50 to 150 range are more useful.

```

100 RESERVE HIMEM-20
110 FOR N=0 TO 19
120   READ J
130   POKE HIMEM+N,J
140 NEXT N
145 REPEAT
150   SOUND HIMEM,100
155 UNTIL NOTKEY$=""
160 PAUSE 20000
165 REPEAT
170   FOR N=1 TO 5
180     READ J
190     SOUND HIMEM,J
200   NEXT N
202   RESTORE
205 UNTIL NOTKEY$=""
210 END
220 DATA 144,100,190,68,205,40,220,40
230 DATA 205,56,190,80,156,104,140,112
240 DATA 132,118,122,0,100,90,80,70,60

```

*Fig. 10.6.* A program which makes use of SOUND. The envelope and waveform are defined by the numbers POKEd into memory, and several different speed numbers are used.

Figure 10.6 shows a SOUND program that includes these instructions. The program POKEs the numbers into reserved memory, and then produces a sound in line 150. Lines 170 to 240 then use another READ...DATA in a FOR...NEXT loop to produce five notes of different pitch using the same set of SOUND numbers in the memory. The main problem in using the SOUND instruction is that it takes a lot of painstaking work to relate the notes that you get to an accepted standard, like the piano scale. There are no hard-and-fast rules, it's all experiment!

The scope for experiment is very wide, though. How about filling several hundred bytes with random numbers, excluding 0, but ending with 0, and then playing this as SOUND? You could then play this with other random numbers used for the delay number that follows the address number. This is computer music with a vengeance, especially if you POKE a different set of numbers into memory for each note! Remember that your program should not

repeat the RESERVE instruction, otherwise you will find that you reserve greater and greater amounts of memory on each run, ending up with no available memory left!

## Spring-clean

Now for some tidying up. In writing about a new computer so full of new instructions and new ideas as the Lynx is, it's certain that there will be instructions that will be missed out. This might be because they were too complicated to explain at the time, or because they didn't appear to fit in anywhere, or because they were so completely explained in the manual, that no more is needed. We can remedy some of these omissions now, because you are no longer a beginner by the time you have worked through to this stage.

Only some of the omissions can be remedied at this stage, however. There simply isn't space to do justice to the many instructions that deal with machine code, because to handle them with confidence, you need to be proficient in Z-80 machine code and the use of hexadecimal numbers. That's a set of topics that needs at least one complete book to itself. All we have space for here is a few BASIC instructions that we missed.

One of these is CFR. This allows you to change the flash rate of the cursor, and it uses a number to set the speed. This number can be any value in the limits 1 to 65535. The effect of CFR 1 is to cause such rapid flashing that you can't really be sure that the cursor is not a steady one. This can have the effect of making the cursor look like an amalgamation of two characters (see CCHAR, later). CFR, on the other hand, is so slow that you could fall asleep waiting for it to change. Take your pick – the normal value is around 5000.

CCHAR is a fascinating instruction which is followed by a number. This number is made out of two ASCII codes, and these codes are the codes of the characters that cause the flash. The normal flashing cursor is created by printing alternately a block and a space, but you can have an asterisk and a space, or an asterisk and a question mark, or any other characters, including user-defined characters, that you like (how about a star and garter?). If you want a steady, non-flashing cursor, you can achieve this by using the same character twice.

Having decided what you want to use, you take the two ASCII codes, multiply the code for one character by 256, and then add the code for the other character. Now type CCHAR followed by this



number, press RETURN, and you have a new cursor! It's very effective in programs, as Fig. 10.7 illustrates.

```
100 CLS
110 PRINT
120 PAUSE 10000
130 CCHAR 32126
140 CFR 100
145 PRINT "TYPE CONT TO CONTINUE"
150 STOP
160 CCHAR 31611
170 PRINT "Now type something!"
180 INPUT A$
190 PRINT A$
200 END
```

*Fig. 10.7.* A cursor-changing program.

The instruction **RANDOM** is a 're-seeding' instruction. The random number generator which produces the numbers for **RND** and **RAND** instructions operates by applying a formula to a 'seed' number – the 'seed' meaning any number that can be used for a starting number in the formula. The numbers that are produced by a formula can never be truly random, and it's possible to see a sequence of 'random' numbers being repeated unless **RANDOM** is used between runs of generating numbers. Other computers use the word **RANDOMIZE** for this same action.

**APPEND** is a cassette system command which allows you to add one program to another. Normally when a program is loaded from tape (or disk), it starts by deleting any program that is already stored. By using **APPEND**, with a filename in quotes following the command word, a program can be loaded in and added to one which is already in the memory. This addition will behave correctly only if the second program which is appended has line numbers that are higher than the line numbers of the first program. If any line numbers coincide, then the appended lines will replace the original lines of the same number. Another useful cassette command is **VERIFY**. If you have just saved a program on tape, you can rewind the tape, type **VERIFY** with the filename in quotes, and then press RETURN and let the tape run again with the **PLAY** key pressed this time. Each piece of data replayed from the tape will be compared with the original version in the memory. If there is any difference between these versions, then the recording has been faulty, and an error message is given. This is particularly useful if you are working with **TAPE 4** or **TAPE 5** speeds.

## Debugging

A bug is a fault in a program. The Lynx will detect and reject items like syntax errors in lines as you enter them, so that any faults that remain in programs are likely to be the result of faulty design rather than cross-fingered typing. Many errors of this type will, in turn, be detected by the Lynx as the program runs, producing error messages that lead you to the source of the trouble. Items like 'Out of Data', faulty 'Dimensioning', 'REPEAT without UNTIL', and so on, will be sorted out as you get the messages in the course of testing a program.

The real problems arise when a program runs, produces no error messages, but gives incorrect results. To start with, you have to know that the results *are* incorrect. This implies that you should test your programs with data for which you already know the correct answers. There's no point in writing an income tax program and testing it with this year's figures. If you test it with figures that have already been tried and approved, then you have a check on how well your program copes with a set of genuine figures. At an earlier stage, it's often better to make up a very simple set of figures which can easily be checked by hand and then tried.

The Lynx offers an excellent range of debugging aids – perhaps we should call them feline insecticides. One of them is STOP. A STOP instruction placed anywhere in your program will cause it to halt at that point, with the message: 'Stopped in line.... (giving the line number)'. You can then use direct PRINT commands to print out values of variables to see what the machine has done with them. By typing CONT (the RETURN) you can get the machine to continue as if nothing had happened. You can even alter the values of variables while the machine has stopped, and it will continue when you use CONT, but using the new values. If you alter any lines of the program, however, in any way, the program cannot be continued, and all the variable values will be wiped. The STOP method of debugging is particularly useful if you think that something is going wrong in a loop.

TRACE ON is another command that is useful for debugging. When TRACE ON has been typed and RETURN pressed, each line number will be printed on the screen as the line is executed. If the program is branching in an unexpected way at some IF step, TRACE ON will reveal this. A common problem is when an ELSE instruction has been used after several IF lines – the ELSE always refers to the immediately preceding IF, not to the earlier ones. The

TRACE action can be turned off by typing TRACE OFF and pressing the RETURN key.

None of the conventional debugging commands is very useful if you have a fault in a graphics drawing program, particularly when the drawing is executed fairly rapidly. Lynx offers a very useful (and unique) SPEED command which will alter the rate at which a program is executed. The normal setting is SPEED 0, which is set automatically when Lynx is switched on, but by typing larger numbers (up to 255) following SPEED, you can slow down execution – and everything else. This is particularly useful for looking at how a graphics display develops, but remember to return to SPEED 0 before you LIST!

## **Final words**

We have come a long way since we started the topic of Lynx programming. Even now, we have delved only skin-deep into the capabilities of the machine because its BASIC is only a starting point. The special fascination of Lynx lies in the use of machine code, for which it is particularly well designed. As we said earlier, there isn't room in one book to do justice to the subject of machine code, which is a much more specialised form of programming language. The BASIC word EXT is said in the manual to offer 'extensions to BASIC', but the use of this on my Lynx produced a message 'not yet implemented'. This message means simply that you have not placed suitable machine code in the memory. Those of us who cut their computing teeth on Nascom or TRS-80 machines and who programmed these machines in Z-80 code, will find themselves very much at home with the Lynx.

At all events, you are no longer a beginner by the time you read this – assuming that you started from the beginning! Enjoy your Lynx, and keep an eye open for the new developments that will almost certainly be coming your way.

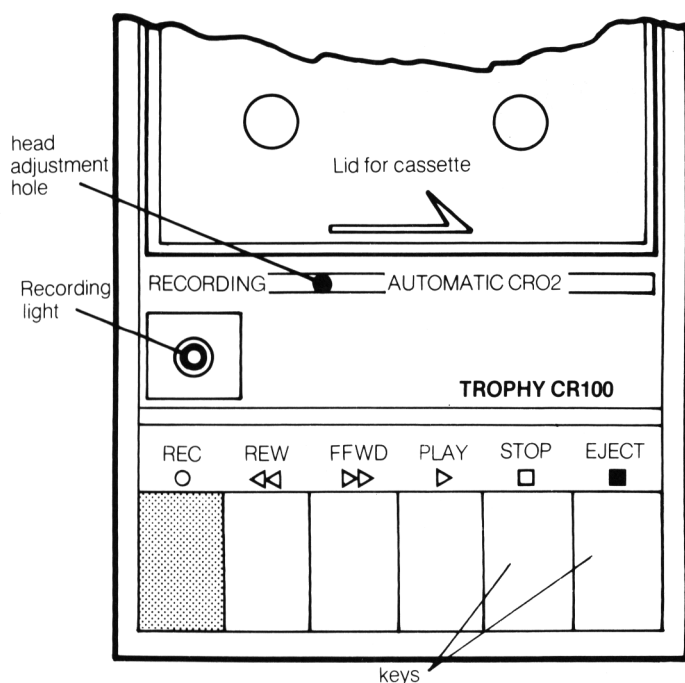
## Appendix A

# Cassette Loading Problems

Cassette recorders, like open-reel recorders, work on the principle of pulling a tape (plastic-coated with iron oxide) past a tapehead, which is a miniature electromagnet. The important part of any tapehead is the 'gap', a tiny slit in the metal, too fine to see except under a microscope. This slit should be tilted so that it is at  $90^\circ$  to the edge of the tape, but this angle is often quite a long way from  $90^\circ$  even when the recorder is new. A poorly set up tapehead will make it difficult to load programs that have been recorded on correctly set up equipment, though you will usually be able to load tapes that you have recorded using the same machine. *Never* touch the tapehead with anything made of metal. You can, however, set up the head to the correct angle using the scheme outlined here. Once you have set up the head correctly, there should be no need to change the setting again unless an odd tape refuses to load. Check before you re-set the head, however, that the trouble is not caused by dirt. Buy a tapehead cleaning kit, and use it as per the instructions.

### Head alignment

You have to start by looking for the head-adjusting screw. The Trophy CR100, for example, has a small hole drilled in the top of the case, just under the cassette lid (Fig. A1). If you insert a jeweller's screwdriver through this hole when a tape is playing, the screwdriver can be engaged in the adjusting screw. First of all, find a cassette which refuses to load. You can obtain control over the cassette motor to allow the sound to be heard by pulling out the remote control jack plug. We will have to remove the EAR plug also to hear the sound. Put the cassette in and play it. Listen to the note. A cassette that refuses to load will usually give a muffled sound, and you have to adjust the head-adjusting screw until this sound is sharp.



*Fig. A1.* The adjustment hole for the head of the CR100 cassette recorder.

You may have to turn the screw either clockwise or anti-clockwise. If a half turn in one direction does not sharpen the sound, or makes it more muffled, then turn in the other direction. You should be able to reach a spot where the sound is quite definitely sharper, more brilliant and with lots of treble. Shut off, restore normal operation, and try to load the cassette. You should now find that it loads. I have never found a cassette that did not respond to this treatment, unless it had been partly erased by being near a magnet. One other failure was due to a notorious computer which did not record its own cassettes correctly!

Not all recorders have the adjusting screw so well placed as that of the CR100, and you may have to remove the cassette lid to reach the screw. In one case (a very cheap recorder) I had to drill a hole for myself, but this is unusual. The higher the quality of the recorder, the more likely that the adjustment will be easy to reach!

# Appendix B

## ASCII Codes

32	,	44	8	56	D	68
! 33	-	45	9	57	E	69
" 34	.	46	:	58	F	70
# 35	/	47	;	59	G	71
\$ 36	ø	48	<	60	H	72
% 37	1	49	=	61	I	73
& 38	2	50	>	62	J	74
' 39	3	51	?	63	K	75
( 40	4	52	@	64	L	76
) 41	5	53	A	65	M	77
* 42	6	54	B	66	N	78
+ 43	7	55	C	67	O	79
P 80	\	92	h	104	t	116
Q 81	]	93	i	105	u	117
R 82	€	94	j	106	v	118
S 83	_	95	k	107	w	119
T 84	£	96	l	108	x	120
U 85	a	97	m	109	y	121
V 86	b	98	n	110	z	122
W 87	c	99	o	111	→	123
X 88	d	100	p	112	←	124
Y 89	e	101	q	113	↑	125
Z 90	f	102	r	114	↓	126
[ 91	g	103	s	115	⇒	127

## Appendix C

# Editing

Editing means making corrections to a line, or lines, of a program. The simplest form of correction is the deletion of a character by means of the DEL key. You can, while you are typing a line, use the left and right arrow keys to move the cursor along the line. Wherever the cursor is placed, you can insert a new character by typing one, or you can delete the character to the left by using the DEL key.

The process that we call editing, however, usually refers to changing a line that has already been entered into the memory by pressing the RETURN key. If the line has *just* been entered, it can be edited by pressing CTRL Q, then RETURN. This will place the cursor at the start of the line. If the line was entered earlier, then type CTRL E and press return. You will get message: 'line number?'. You must then type the line number of the line that you want to edit, and press RETURN again. Once more this places the line on the screen with the cursor at the start of the line.

Editing is now carried out using the cursor. The left and right arrow keys will move the cursor to the part of the line that you want to edit. If this is a long distance from the cursor, then the down-arrow key will move the cursor to the end of a line; the up-arrow key will return it to the beginning. When the cursor is positioned where you want it, pressing the DEL key will delete the character to the left of the cursor, and typing a new character will insert that character at the cursor position. This will cause all the remaining characters in the line to move to the right. This is a comparatively slow process, and you may have to type deliberately slowly unless your typing speed is already slow! Once editing is carried out to your satisfaction on that line, you can re-enter the new version into memory by pressing the RETURN key. Check that it looks correct by listing that line again.

## Appendix D

# Magazine and User Groups

Once you have begun to explore the possibilities of computing for yourself, the help of magazines and user groups becomes extremely valuable. Computing is a fast-moving business, so that only magazines can appear often enough to keep you up to date with developments. It's in the magazines that you will see the advertisements for hardware (like printers and disk drives) and software (like programs that let you use printers and disk drives!). It's in the magazines that you will also see reviews of equipment, programs and books that can guide you to your next steps in computing.

User groups are much more specialised. A user group consists of a number of people who have bought one make of computer and are determined to get as much out of it as possible. They find out flaws that the manufacturers may not have suspected, find ways round these flaws, and press the manufacturers for changes. More important, they can be of immense help to the novice. Every member of a user group, when you first join, seems to be an expert who is willing to devote endless time to helping you. When it's your turn to act as expert to some newcomer, you find that you derive as much pleasure (and you often learn a lot as well) from helping as from being helped. I have many good memories of the friendly assistance that I had from the TRS-80 User Group when I started microcomputing – the fact that I first learned programming in 1962 didn't help me to understand the TRS-80!

At the time of writing, there weren't many Lynxes skulking around the country, and no user group had been formed. There are dozens of magazines, though, and some names may be helpful to you. *Personal Computing World* is the grand-daddy of all U.K. computing magazines aimed at the small computer user. It contains a wealth of advertisements, and its articles are of particular interest once you are past the 'raw beginner' stage. The features such as 'T.



*J.'s Workshop* and *'Subset'* are particularly useful. *T. J.'s Workshop* deals with solutions to problems in BASIC, and with hints to help users of specific computers. *Subset* is aimed at machine language programmers, and contains so many subroutines for the Z-80 that it amounts almost to a Nascom/TRS-80/MZ-80/Spectrum/Lynx benefit page.

*Your Computer* is aimed more at the programmer whose interests are in games and other leisure programming. It also carries a good range of advertisements, and was one of the first monthly magazines to publish a review of the Lynx. On the other hand, if you are interested in how computers work and how they can be harnessed to other devices, then *Computing Today* and *Electronics and Computing Monthly* are very much aimed at you.

There is such a huge variety of magazines that you should really sample one of each until you find one (or more) that will keep your interest in every issue. Once you find it – subscribe. The demand for computing magazines is such that you can't rely on finding a magazine casually in a stationer's shop unless you search daily. There's nothing worse than missing one part of a series just because you didn't turn up on publication day!

## Appendix E

# Mystery Corner

Try this program:

```
1Ø FOR N = Ø TO 3
2Ø PRINT CHR$(PEEK(27+N));
3ØNEXT N
```

RUN it. Is there a prize somewhere for guessing who this is? Is there a golden hare buried somewhere, or is it golden hair?

# Appendix F

## Drawing Circles

The Lynx lacks a CIRCLE command, but either of the following procedures will draw circles – you will have to decide for yourself which is more suited to your purposes. The procedures are called in the usual way, giving the x and y co-ordinates of the centre of the circle, and the radius value r.

### Circle procedure 1

```
1000 DEFPROCcirc(x,y,r)
1010 MOVE x,y+r
1020 FOR n = 0 TO 360 STEP 18
1025 REM make step smaller for better
appearance of circle
1030 DRAW x+r*SIN(RAD(n)),y+r*COS(RAD(n))
1040 NEXT n
1050 ENDPROC
```

### Circle procedure 2

```
2000 DEFPROCcerc(x,y,r)
2010 MOVE x-r,y
2020 FOR J=x-r TO x+r
2030 LET p=SQR(r**2-(J-x)**2)
2040 DOT J,y+p
2050 DOT J,y-p
2060 NEXT J
2070 ENDPROC
```

# Glossary

*Acoustic coupler:* a method of sending program information down telephone lines by converting to sound. The telephone attaches to the loudspeaker and the microphone of the coupler, and is held firmly against them. Only low baud rates can be used, but it avoids direct connection to the telephone lines, frowned on by British Telecom.

*Applications software:* programs that make the micro do a particular job for you; what you are applying the microcomputer to.

*Assembler:* a program or hardware circuit which permits machine code to be written making use of mnemonics, such as LDA, STA, rather than directly in binary number codes. The use of an assembler makes machine code very much easier to write.

*Assembly language:* the set of mnemonics which is used to write programs for a microprocessor, using an assembler. Each instruction consists of an operator, which defines an action, and an operand, which defines the byte or bytes on which the action is carried out.

*BASIC:* stands for Beginners All-purpose Instruction Code; the most common computer language found on micros.

*Baud rate:* rate of transfer of information, named after the eminent Belgian telegraph engineer, Baudot. There is no simple equivalent between baud rate and number of bytes transferred per second, but a ratio of 10:1 is often assumed, so that 300 baud is approximately 30 bytes per second.

*Benchmark:* a standard set of instructions, designed to measure the speed at which a processor operates.

*Binary:* a number code which uses only the digits 0 and 1. Computers in general use simple binary (8-4-2-1) code, but other varieties, such as grey code are used for machine control.

*Bit:* a binary digit; this is the actual electrical signal that is used in a computer – either a 0 or a 1.

*Buffers:* isolating stages in a signal line. The presence of a buffer prevents a signal source from being overloaded when other circuits are connected. A buffer may also be arranged so as to switch the lines on and off as needed.

*Bus:* a set of connecting lines in a microprocessor or computer system. The name bus (omnibus = for all) is used because the lines interconnect all the units of the system, and allow signals to be passed between any pair of units that are connected to the bus.

*Byte:* roughly equal to a letter or number and composed of several bits; each bit being 0 or 1 builds up the code for that character.

*Cassette:* a container for magnetic tape, complete with reels and pressure pads. Audio cassettes are commonly used for small computers, but specially made digital cassettes are used for larger computers. Miniature digital cassettes are capable of storage of large amounts of information in small spaces.

*Centronics interface:* a special purpose printer interface developed by the Centronics Company. A parallel interface, it also allows the computer to control the printer, whilst still sending it data.

*Chip:* see Processor.

*COBOL:* standing for COMmon Business Orientated Language, it has been specially developed to meet the needs of the business user.

*Compiler:* a special program that translates your program into a form that the computer understands, creating a second machine language program; it is this second program that is used to do the actual work.

*CP/M:* stands for Control Program Microcomputer; virtually a standard operating system on low cost business micros.

*Daisywheel printer:* a high quality printer that uses a print element with all the letters arranged on spokes – like a daisy!

*Data:* this is your information that is held on the computer.

*Database:* a collection of several different types of information, such as names and addresses and accounts, which can be combined to create other types of data – such as outstanding balances due reminders.

*Disk:* see Floppy disk.

*Disk drive:* this device reads data off, or writes it to your disks.

*Dot matrix printer:* a type of printer that builds up letters and numbers from a series of dots printed on the paper.

*Dynamic memory:* a system of memory in which information is stored as electric charges on capacitor plates. Because the charge leaks away, the charged plates have to be 'refreshed' to avoid loss

of stored signal. Some microprocessors, notably the Z-80, provide for refreshing signals on the main chip.

*EPROM: Erasable PROM.* A re-programmable memory which, once programmed, will hold information until re-programmed. Since the stored information is non-volatile (not lost when power is switched off) and cannot be altered by the normal action of the microprocessor, an EPROM can be used in a computer to hold a permanent program, which is available when the machine is switched on.

*Ethernet:* this is a network based on a single cable, into which micros are spliced along its length. No controller is used.

*Firmware:* a program stored in ROM or EPROM rather than on disk or tape.

*Floppy disk:* a metal-oxide covered plastic disk inside a cardboard envelope that is used to hold programs and data on a microcomputer. (See Track, Sector, Soft-sectoring, Hard-sectoring.)

*GIGO:* stands for Garbage In – Garbage Out: meaning that if you do not give the computer the correct information, you can hardly expect it to give you the correct end result.

*Graphics:* used to describe pictures or graphs which can be created using special programs designed to control the screen or printer.

*Handholding:* the period during which your supplier will help you get up and running, free.

*Handshake:* a form of synchronisation system that is used when data is transferred. A handshake signal can be used to indicate that data can be sent, usually from a computer to another unit such as a printer or another computer.

*Hard disk:* see Winchester disk.

*Hard-sectoring:* a disk which has a set of index holes permanently punched in it is said to be 'hard-sectored'. Such disks are seldom used for microcomputers – see soft-sectoring.

*Hardware:* all the electronics and mechanical devices that make up a micro system are hardware; to distinguish them from programs – which are software.

*Hex:* short for hexadecimal, a scale of sixteen counting digits. The advantage of using such a scale, generally in conjunction with assembly language, is that one byte can be expressed as two hex digits. Conversion between hex and binary is also very simple.

*Interface:* the physical shape of the plug and socket at each end of a cable, and the type and voltage of the current passing over it, that is used to link a printer or screen, etc. to the computer.

*Interpreter:* this translates your programs one line at a time, every time you use that program.

*Joystick:* a device as used on aircraft, except that on micros, it moves a pointer over the screen. Some have firing buttons on the top especially for computer games.

*Keyboard:* what you use to type in programs and data, or to respond to prompts on the screen.

*Language:* used when referring to the computer language that a program is written in.

*Letter quality printer:* used to describe a printer that can produce print comparable to a typewriter.

*Loop:* a sequence of commands within a program that are intended to be repeated over and over again.

*Machine code:* a set of binary numbers that act as instructions and data codes for a microprocessor. This is the programming language for the microprocessor itself, and programs written in machine code will run as fast as the clock rate of the microprocessor permits. The use of machine code also permits actions which cannot be achieved using BASIC.

*Mainframe:* a very large computer that can hold complex programs, and handle lots of data at once. It is made up of several special types of processor, and can cope with many users at once.

*Memory:* this is the area inside the computer that is used to hold your programs.

*Menu:* a series of options that are shown on the screen, from which you select the appropriate function, according to what you want the program to do.

*Microprocessor:* see Processor.

*Minicomputer:* a computer designed to cope with several people on the system at once, using very fast, high capacity disk drives for data storage.

*Minifloppy disk:* see Floppy disk.

*Mnemonics:* shortened instruction words that are used in assembly language to describe microprocessor action.

*Modem:* this device is used to allow a micro to send or receive information over ordinary telephone lines.

*Multi-user micro:* exactly what it says: a micro that can be used by more than one person.

*Network:* here, several micros are connected by cables, in such a way that each micro can get at information held anywhere else on the system.

*OEM:* a term used in the industry for a supplier who takes a micro

from the makers, and adds his own software package to sell it as a complete system. In some cases, he may even 'own-badge' the product.

*Operating system:* all computers have special software that comes with that machine, and takes care of all the commonly needed jobs such as reading from disks, or sending information to the printer. Your program only has to specify what it wants to send or get, and from where, and the operating system will take over from there.

*Parallel interface:* used to describe an interface that can cope with a letter or number as a complete byte – instead of having to handle it bit by bit.

*PEEK:* a BASIC instruction which finds the contents of memory. The syntax is that an address number follows PEEK, enclosed in brackets, and the number that is returned is the number stored at that memory address.

*Peripheral:* any device that is connected by a cable to the processor – such as a printer or a VDU.

*Pixel:* dots of light on the screen that are used to make letters or numbers. The more of them, the crisper the character.

*POKE:* a BASIC instruction that changes the contents of memory. The syntax is POKE address, and the data number (which must be between 0 and 255) will be placed in the memory address. Use of POKE requires some knowledge of the memory structure of the computer.

*Processor:* at the heart of any computer, this is the unit that holds your program instructions in memory, and controls all the devices linked to it through the operating system.

*Program:* the set of instructions written in a computer language that spell out logically what you want the computer to do.

*PROM:* Programmable Read-Only Memory. A form of non-volatile memory which can be programmed by the user, so that programs can be stored permanently. Used for cartridges.

*Protocol:* the format in which data and program instructions are passed over an interface.

*QWERTY keyboard:* all this means, is that the micro or VDU has a typewriter-style keyboard.

*RAM:* stands for Random Access Memory – or memory for short.

*Read/Write Head:* the device that picks data off the surface of a disk, and puts fresh information on it.

*ROM:* this is Read Only Memory, and is used to hold both the operating system and a computer language on some home



computers. You cannot use this type of memory for your programs.

*RS232*: a commonly used interface standard.

*SI00*: one of the best-known and most widely used bus systems for connecting a computer to its peripherals. The use of a bus of this type is particularly appropriate when the computer comprises several boards which have to be interconnected, and which are replaceable.

*Sector*: a portion of a disk track. The circular track is divided up into a number of such sectors, each of which can be used to store information. The division into sectors may be hard or soft (qv).

*Serial interface*: where information is sent over a cable a bit at a time. Each byte representing a number or letter, is built up in stages.

*Soft-sectoring*: a system of sectoring which is achieved by using only one sector index hole. The computer then marks out sectors on the disk, using signals, in a 'formatting' process.

*Software*: this refers to programs – both the applications programs you have bought or written, plus languages and the micro's operating system.

*Star network*: a mixture of multi-user micro and network proper, each machine has its own cable to a controller at the hub of the star – which usually doubles as a Winchester disk drive.

*Static memory*: a type of memory in which storage is achieved in minute electrical switches, called flip-flops. Unlike dynamic memory, the data will be retained for as long as power is applied and no refreshing is needed, but more power is consumed.

*System software*: the term used to describe those programs that are specific to the microcomputer. Included here are languages, the operating system, and various programs to do 'housekeeping' chores such as copying disks and deletions.

*Tailoring*: where a standard software package is altered to more closely meet your needs.

*Track*: a complete circular groove in a magnetic disk. Disks are typically 40 or 80 track, and the track is subdivided into sectors for the purposes of controlling the allocation of storage space.

*Turnkey system*: a matched package of applications software and micro system, that is designed to be used with minimal alteration. Some of these are meant to be started up by the turn of a key – literally!

*VDU*: standing for Visual Display Unit, this is a combination of keyboard and screen.

*Volatile*: a system of memory in which data is lost when power is

switched off. Most computer memory systems of RAM are volatile. When a volatile memory is switched on again, each memory cell will switch on at random, so that meaningless bytes (garbage) will be stored.

*Winchester disk*: sealed metal disk drive, that spins at high speed, and can carry large amounts of information, as well as being very quick at reading or writing data.

# Index

@ instruction, 18

ABS, 42

absolute co-ordinates, 106

adaptors, 9

aerial adaptor, 7

alien figures, 113

amplitude and frequency, 121

animated drawing, 111

antilogarithms, 44

APPEND, 66, 127

arithmetic signs, 14

array, 80

ASC, 55

ASCII code, 37, 132

assigning, 21

attack, 121

attention-getter, 61

AUTO, 17

bargain tapes, 8

BASIC, 1

BEEP, 100

Beep, 4

blank line, 19

blank string, 64

brackets, 42

breaking a loop, 27

calculator work, 14

cassette problems, 129

cassette lead, 8

cassette recorder, 8

cassettes, 8

CCHAR, 126

Centring, 18

CFR, 126

character block, 91

character-planning grid, 114

checking entries, 74

values, 74

CHR\$, 57

circles, 137

clear line code, 64

clear screen, 16

coarse graphics, 88

code actions, 59

colour, 93

colour mixtures, 93

colour monitor, 1

colour protection, 96

colour text and graphics, 95

colour TV, 6

comma in INPUT, 24

commas, 17

comparison conditions, 37

concatenation, 51

conditions, 36

CONT, 128

co-ordinates, 104

core program, 75

correct order, 38

countdown, 30

counting, 26

crash-through, 67

crashing through, 29

cursor, 10, 104

damage, 8

data filing, 87

day-of-the-week, 33

debugging, 128

decay, 121

defining procedure, 71

DEL key, 10

deleting cursor, 106

delta-wing shape, 115

dial tuning, 4

DIM, 22

dimensioning, 22

array, 81

string array, 85

direct command, 14  
 DIV, 48  
 DOT, 108  
 double-height letters, 63  
 DPOKE, 117  
 DRAW, 106  
 drawing a shape, 106  
 drawing circles, 137

EAR socket, 9  
 editing, 16, 32, 133  
 envelope, 121  
 errors in BASIC, 66  
 ESC key, 12  
 experiments in sound, 125  
 exponentiation, 42  
 expression, 41  
   in GOSUB, 68  
 EXT, 128

factorial, 47  
 fields, 61  
 filename, 11  
 finding number, 84  
 flag variable, 43  
 flashing asterisk, 65  
 formal parameter, 71  
 FORTH, 112  
 FRACT, 48  
 fuse, mains, 3

GET\$, 32, 64  
 GOSUB, 66  
 GOTO, 27  
 GRAPHIC, 113  
 graphics, 88  
   code number, 92  
   in DATA line, 91  
   symbols, 89  
 graphs, 46, 104  
   programs, 109  
 Greek characters, 112

head adjustment, 129  
 hexapuss, 118  
 high resolution, 88  
   graphics, 104  
 HIMEM, 116  
 home position, 105

IF instruction, 28  
 indenting, 30  
 INF, 50  
 INK, 94  
 INPUT, 20

INT, 41  
 integer part, 41  
 invisible cursor, 62

jack plugs, 8  
 joining strings, 51

keyboard, 12  
 KEY\$, 64

label in GOSUB, 69  
 label-number, 80  
 leader, tape, 9  
 LEN, 52-3  
 LETTER, 118  
 line number, 15  
 line-space codes, 60  
 line up numbers, 53  
 lining up decimals, 56  
 LIST, 16  
 logarithms, 44  
 low resolution, 88  
 lower-case, 13

magazines, 134  
 main program, 75  
 mains supplies, 2  
 marks program, 79  
 MIC socket, 9  
 Microvitec, 1  
 MOD, 48  
 mode number, 110  
 monitor, 1  
 MOVE, 105  
 mugtrap, 69  
 multiple characters, 118  
 music for Lynx, 102  
 musical scale, 100  
 mystery corner, 136

nested procedures, 76  
 nesting, 30  
 NEW, 17  
 number accuracy, 49  
 number functions, 39  
 number of cycles, 101

on/off switch, 4  
 ordered list, 84  
 organising programs, 66  
 other loops, 34  
 overwriting, 63

packing characters into string, 93  
 PAPER, 94

passing a variable, 70  
 passing numbers, 72  
 passing variable back, 72  
 PAUSE, 30  
 peculiar effects, 7  
 PEEK, 113  
 PI ( $\pi$ ), 48  
 pixels, 104  
 planning grid, LR, 92  
 PLOT, 110  
 pointer for DATA, 32  
 POKE, 113  
 pound sign, 41  
 power pack, 2  
 power plug, 2  
 precedence, 42  
 primary colours, 93  
 print modifiers, 16  
 PRINT, 14  
 PRINT@ numbers, 19  
 printing a pattern, 117  
 PROC, 71  
 program, 15  
     design, 66, 73  
 prompt, 23, 73  
 PROTECT, 96

Quote marks, 10

RAND, 40  
 RANDOM, 127  
 random number, 39  
 READ and DATA, 31  
 recording speeds, 12  
     test, 9  
 redimensioned array, 81  
 relative co-ordinates, 110  
 release, 122  
 REM socket, 9  
 remote plug leads, 9  
 RENUM, 66  
 REPEAT...UNTIL, 34, 35  
 replay, 11  
 RESERVE, 116  
 RESTORE, 32  
 RETURN instruction, 66  
 RETURN key, 14  
 RETURN without GOSUB, 67  
 ROUND, 49  
 running total, 27

SAVE, 10  
 scientific form, 49  
 semicolon, 16  
 SGN, 43

shape on screen, 90  
 SHIFT key, 10  
 single statement lines, 25  
 skeleton program, 73  
 sorting numbers, 82  
 sound, 100  
     envelope, 124  
     planning chart, 122  
 SOUND, 120  
 special keys, 13  
 SPEED, 128  
 standard form, 49  
 STEP, 30  
 STOP, 128  
 STR\$, 52  
 string, 17  
     arrays, 85  
     length, 21  
     slicing, 54  
     variable, 21  
 structure, 66  
 subroutine, 66  
 subscript, 61, 80  
 superscripts, 60  
 sustain, 121  
 SWAP, 38  
 syntax error, 23

TAB instruction, 18  
 test program, 10  
 testing entry, 28  
 TEXT, 16, 97  
 text mode, 88  
 three-pin plug, 3  
 time delay loop, 29  
 tone control, 12  
 top-down design, 72  
 TRACE, 128  
 tracing a shape, 107  
 TRAIL, 50  
 trigonometry, 44  
 Trophy CR-100, 8  
 tuning defects, 6  
 tuning, TV, 4  
 TV cable, 3  
     channels, 4  
     connections, 4  
     receiver, 1  
     tuning, 4  
 type mismatch, 32

undefined variable, 24  
 UPC\$, 58  
 upper-case, 13  
     conversion, 25

## 148 *Index*

- converter, 57
- use of procedure, 70
- User-defined characters, 113
  - graphics, 88
- user groups, 134
- using END, 29
  - LET, 25
    - a subroutine, 67
  - arrays, 81
  - string array, 86
- VAL, 51
- Variable name, 20, 22
- VDU, 58

- VERIFY, 127
- visual display unit, 58
- volume, 101
- volume control, 12
- warbling note, 103
- warning note, 102
- wavelength number, 100
- waveshapes, 121
- WHILE...WEND, 35
- WINDOW, 97
- window numbers, 98
- wiping a window, 99



## HOW TO MASTER THE LYNX!



The Lynx has attracted widespread interest because of its remarkable value for money and easy expandability. It is designed to be interfaced to peripherals made by other manufacturers, so that users are not necessarily restricted to buying from (or waiting for) a single range of equipment. It offers superb facilities, including eventual CP/M compatibility to open the way to a comprehensive selection of ready-made software.

Aimed at all users, this book starts at the very beginning with how to set up the machine. It then goes on to guide you step by step until you become sufficiently expert to write your own programs and start using the machine creatively for your own special purposes.

Very many useful programs are included and you will continue to find the book useful as a handy reference even after you have mastered all the techniques.

### *The Author*

Ian Sinclair is a well-known and regular contributor to journals such as *Personal Computer World*, *Computing Today*, *Electronics and Computing Monthly*, *Hobby Electronics* and *Electronics Today International*. He has written some forty books on aspects of electronics and computing mainly aimed at the beginner.

Photograph of the Lynx microcomputer (Computers Ltd.)  
by courtesy of Lang Communications Ltd.

Photograph of lynx by Jane Burton, by permission  
of Bruce Coleman Ltd.



STANLEY  
XCO  
MPTING

GRANADA